Swedish Institute of Computer Science
Doctoral Thesis
SICS Dissertation Series 47

Programming Memory-Constrained Networked Embedded Systems

Adam Dunkels

February 2007

Swedish Institute of Computer Science
Stockholm, Sweden

Copyright © Adam Dunkels, 2007
ISRN SICS-D–47–SE
SICS Dissertation Series 47
ISSN 1101-1335
Printed by Arkitektkopia, Västerås, Sweden

Abstract

Ten years after the Internet revolution are we standing on the brink of another revolution: networked embedded systems that connect the physical world with the computers, enabling new applications ranging from environmental monitoring and wildlife tracking to improvements in health care and medicine.

Only 2% of all microprocessors that are sold today are used in PCs; the remaining 98% of all microprocessors are used in embedded systems. The microprocessors used in embedded systems have much smaller amounts of memory than PC computers. An embedded system may have as little has a few hundred bytes of memory, which is thousands of millions times less than the memory in a modern PC. The memory constraints make programming embedded systems a challenge.

This thesis focus on three topics pertaining to programming memory-constrained networked embedded systems: the use of the TCP/IP protocol suite even in memory-constrained networked embedded systems; simplifying event-driven programming of memory-constrained systems; and dynamic loading of program modules in an operating system for memory-constrained devices. I show that the TCP/IP protocol stack can, contrary to previous belief, be used in memory-constrained embedded systems but that a small implementation has a lower network throughput. I present a novel programming mechanism called protothreads that is intended to replace state machine-based event-driven programs. Protothreads provide a conditional blocked wait mechanism on top of event-driven systems with a much smaller memory overhead than full multithreading; each protothread requires only two bytes of memory. I show that protothreads significantly reduce the complexity of event-driven programming for memory-constrained systems. Of seven state machine-based programs rewritten with protothreads, almost all explicit states and state transitions could be removed. Protothreads also reduced the number of lines of code with 31% on the average. The execution time overhead of protothreads is on the order

of a few processor cycles which is small enough to make protothreads usable even in time-critical programs. Finally, I show that dynamic linking of native code in standard ELF object code format is doable and feasible for wireless sensor networks by implementing a dynamic loading and linking mechanism for my Contiki operating system. I measure and quantify the energy consumption of the dynamic linker and compare the energy consumption of native code with that of virtual machine code for two virtual machines, including the Java virtual machine. The results show that the energy overhead of dynamic linking of ELF files mainly is due to the ELF file format and not due to the dynamic linking mechanism as such. The results also suggest that combinations of native code and virtual machine code are more energy efficient than pure native code or pure virtual machine code.

The impact of the research in this thesis has been and continues to be large. The software I have developed as part of this thesis, lwIP, uIP, protothreads, and Contiki, is currently used by hundreds of companies in embedded devices in such diverse systems as car engines, oil boring equipment, satellites, and container security systems. The software is also used both in academic research projects and in university project courses on embedded systems throughout the world. Articles have been written, by others, in professional embedded software developer magazines about the software developed as part of this thesis. The papers in this thesis are used as required reading in advanced university courses on networked embedded systems and wireless sensor networks.

Sammanfattning

Tio år efter Internet-revolutionen står vi nu inför nästa revolution: kommunicerande inbyggda system som kopplas ihop med varandra och därigenom möjliggör helt nya tillämpningar inom ett stort antal områden, bland annat sjukvård, miljöövervakning och energimätning.

Endast 2% av alla mikroprocessorer som säljs idag används för att bygga PC-datorer; resterande 98% går till inbyggda system. Det stora flertalet av dessa system har avsevärt mycket mindre minne än en modern PC. Inbygga system har ofta endast ett par hundra bytes minne, att jämföra med de tusentals miljoner bytes minne en modern PC har. Minnesbegränsningarna hos de inbyggda systemen gör dem till en utmaning att programmera.

Avhandlingen behandlar programmering av minnesbegränsade kommunicerande inbyggda system ur tre synvinklar: möjligheten för mycket små minnesbegränsade inbyggda system att kommunicera med hjälp av Internet-protokollen; förenkling av händelsestyrd programmering för minnesbegränsade system; och dynamisk programladdning i ett operativsystem för kommunicerande inbyggda system.

För att ett inbyggt system ska kunna kommnunicera i ett nätverk krävs att systemet kan prata nätverkets språk, nätverksprotokollet. Avhandlingen visar att det är möjligt även för mycket små system att använda Internet-protokollen, TCP/IP, utan att behöva göra avsteg från gällande Internet-standarder. Dock innebär minnesbegränsningarna en avsevärd prestandaminskning. Min programvara lwIP och uIP visar att det är möjligt att koppla ihop mycket enkla inbyggda system, väsentligt mycket mindre än man tidigare trott, med nätverk som använder TCP/IP-protokollen.

Många program för minnesbegränsade inbyggda system bygger på en programmeringsmetod som kallas händelsestyrd programmering. Med händelsestyrd programmering kan man skriva program som kräver mycket lite minne, men programmen blir ofta svåra att både utveckla och underhålla. För

att underlätta programmeringen av sådana system har jag utvecklat en programmeringsteknik som jag kallar protothreads, prototrådar. Protothreads gör händelsestyrda program mindre komplexa utan att nämnvärt öka minnesutnyttjandet och med mycket små prestandaförluster.

Operativsystemet Contiki, som jag har utvecklat under avhandlingsarbetet, kan under drift ladda nya programmoduler, något som inte stöds av andra operativsystem för små inbyggda system. Att kunna ladda programvara under drift underlättar både utveckling av ny programvara och korrigering av felaktig programvara. Contiki visar att det trots resursbegränsningarna är möjligt att ladda programmoduler i standardformatet ELF. Jag kvantifierar energiåtgången både för att ladda program med dynamisk länkning och för att exekvera de laddade programmen, samt jämför energiåtgången med den för motsvarande program skrivet för två virtuell maskiner, bland annat en Java-maskin.

Programvaran har fått mycket stor spridning och används idag av hundratals företag i ett stort antal produkter såsom bilmotorer, satellitsystem, oljeborrar, TV-utrustning och låssystem från företag såsom BMW, NASA och HP. Programvaran används i projektkurser på universitetsnivå världen över. Artiklar i branchtidskrifter har skrivits, av utomstående, om hur man anpassar programvaran för nya mikroprocessorer. Ledande experter inom programvaruutveckling för inbyggda system har ett flertal gånger rekommenderat programvaran i nyhetsbrev. Både forskningsartiklarna i avhandlingen och programvaran används i undervisningen vid universitet världen över.

To Castor, Morgan, Maria, and the little one we have not yet met

Preface

I have always loved programming. When I was a kid, my father sometimes brought home a computer that he used in teaching computer programming to mathematic teacher students at the university. The computer was an ABC80, a Swedish Z80-based computer with a BASIC interpreter and 16 kilobytes of RAM. I learned programming BASIC from modifying my father's programs and by typing in BASIC programs from the manual. The programs were very small and I was never limited by the small amount of memory. A few years later I got my first own computer, a Commodore 64 with 64 kilobytes of RAM. I was so eager to start programming that I had already learned programming the assembly language of its 6510 processor by reading books on the subject before I got the actual computer. Over the years, I wrote a large number of assembly language programs for it and frequently felt that its memory was a limitation. Some six or seven years later I bought my first PC, with a 486 microprocessor and 16 megabytes of RAM. I quickly learned x86 assembly language but never came anywhere near writing a program that used the entire memory of the machine.

In 2000 I did my master's thesis at the Swedish Institute of Computer Science in Kista, Sweden. As part of my thesis I developed a TCP/IP stack, which I named lwIP, for transmitting vital statistics from wireless sensors on ice hockey-players to people in the audience with laptop computers. The wireless sensors were equipped with Mitsubishi M16c CPUs with 20 kilobytes of RAM and 100 kilobytes of ROM. I was almost back where I started!

While this thesis officially was done over the past four years at the Swedish Institute of Computer Science, the real work started almost 20 years earlier.

Adam Dunkels
Stockholm, January 8 2007

Acknowledgements

I first and foremost thank my colleague Thiemo Voigt, who has also been the co-adviser for this thesis, for all the moral support over the past few years, for being the committed person that he is, and for being genuinely fun to work with. Working with this thesis would have been *considerably* less enjoyable if it had not been for Thiemo. Thiemo and I have been working during the final hours before paper submission deadlines, sometimes as late/early as 6 AM in the morning. Thiemo has also gone out of his way to take care of distracting duties, thus allowing me to focus on doing the research for this thesis.

I am grateful to Mats Björkman, my university adviser for this thesis, for being a stimulating person and for the smooth PhD process. Seemingly big problems have always turned into non-problems after a discussion with Mats.

I am also very grateful to the inspiring Juan Alonso. Juan started the DTN/SN project at SICS within which most of the work in this thesis was done. I also would like to thank Henrik Abrahamsson for being a good friend and stimulating discussion partner on subjects ranging from the craft of science and research to cars and culinary culture. I am also very happy to work with the great members of our Networked Embedded Systems Group at SICS: Joakim Eriksson, Niclas Finne, and Fredrik Österlind. An equally skilled and dedicated group of people is very hard to find. Many thanks also to Björn Grönvall for taking a lot of the work of writing project deliverables as well as porting Contiki to new platforms. Thanks also to Sverker Janson, laboratory manager of the Intelligent Systems Laboratory at SICS, for his inspiring leadership and for his support. Many thanks to all the people at SICS for creating a stimulating work environment; Lars Albertsson, Frej Drejhammar, Karl-Filip Faxén, Anders Gunnar, Ali Ghodsi, Kersti Hedman, Janusz Launberg, Ian Marsh, Mikael Nehlsen, Martin Nilsson, L-H Orc Lönn, Tony Nordström, Carlo Pompili, Babak Sadighi, and Karl-Petter Åkesson, just to name a few.

Many thanks to Oliver Schmidt for our cooperation on protothreads and his

porting and maintaining of Contiki, for always being a very sharp discussion partner, and for being a good person to work with.

Thanks also to the great master thesis students with whom I have been involved during this work: Max Loubser, Shujuan Chen, Zhitao He, and Nicolas Tsiftes. Thanks also to Muneeb Ali for his fruitful research visit at SICS.

My thanks also go out to the hundreds of people I have been in contact with regarding my software over the past few years. I have gotten many warming words, good comments on my software, bugfixes and patches, as well as new modules and ports to new architectures. I have gotten so many e-mails that I unfortunately have only been able to answer a fraction of them.

I am also deeply grateful to the people at Luleå University of Technology for teaching me the basic aspects of computer science. Lars-Gunnar Taube for introducing me to the secrets of computing many years ago; Håkan Jonsson for his introduction to the interesting world of functional programming; Leif Kusoffsky for his imperative programming laboratory assignments that taught me how to write virtual machines and how to develop compilers for object-oriented languages; Lennart Andersson for giving me the *extremely* important insight that external and internal data representation need not be the same, when we were instructed to not use a two-dimensional array to represent the spreadsheet data in the VisiCalc-clone we developed as a laboratory assignment; Mikael Degermark and Lars-Åke Larzon for sparkling my interest in computer communications; and Olov Schelén and Mathias Engan for teaching me how to read and review scientific papers.

Thanks also go to my mother Kerstin for being supportive throughout my education and research career, for taking interest in my research work, and for reading and commenting on this thesis. I will also forever be in debt to my late father Andrejs, who taught me the skills of everything from living and laughing to mathematics and music.

Finally, I am extremely fortunate to have been blessed with such a loving family: my wife Maria, our sons Morgan and Castor, and one who we look forward to meet in a few months from now. Maria has supported me throughout the work with this thesis, taken interest in my work, listened to and helped improve my research presentations, and endured all my research ramblings at home.

This work in this thesis is in part supported by VINNOVA, Ericsson, SITI, SSF, the European Commission under the Information Society Technology priority within the 6th Framework Programme, the European Commission's 6th Frame-

work Programme under contract number IST-004536, and the Swedish Energy Agency. Special thanks to Bo Dahlbom and the Swedish Energy Agency for funding the final writing up of this thesis. The Swedish Institute of Computer Science is sponsored by TeliaSonera, Ericsson, SaabTech, FMV, Green Cargo, ABB, and Bombardier Transportation AB.

Included Papers

This thesis consists of a thesis summary and five papers that are all published in peer-reviewed conference and workshop proceedings. Throughout the thesis summary the papers are referred to as Paper A, B, C, D, and E.

Paper A Adam Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (ACM MobiSys 2003)*, San Francisco, USA, May 2003.

Paper B Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE Emnets 2004)*, Tampa, Florida, USA, November 2004.

Paper C Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. Using protothreads for sensor node programming. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN 2005)*, Stockholm, Sweden, June 2005.

Paper D Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (ACM SenSys 2006)*, Boulder, Colorado, USA, November 2006.

Paper E Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (ACM SenSys 2006)*, Boulder, Colorado, USA, November 2006.

Contents

I Thesis Summary **1**

1 Introduction **3**
1.1 Wireless Sensor Networks 4
1.2 Programming Memory-Constrained Embedded Systems . . . 5
1.3 Research Approach and Method 5
1.4 Research Issues 6
1.4.1 TCP/IP for Memory-Constrained Systems 7
1.4.2 Protothreads and Event-Driven Programming 8
1.4.3 Dynamic Module Loading 8
1.5 Thesis Structure 9

2 Scientific Contributions and Impact **11**
2.1 Scientific Contributions 11
2.2 Impact 12

3 Summary of the Papers and Their Contributions **15**
3.1 Paper A: Full TCP/IP for 8-Bit Architectures 16
3.2 Paper B: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors 17
3.3 Paper C: Using Protothreads for Sensor Node Programming . 18
3.4 Paper D: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems 19
3.5 Paper E: Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks 19

4 Related Work **21**
4.1 Small TCP/IP Implementations 21

4.2 Operating Systems for Wireless Sensor Networks 22
4.3 Programming Models for Wireless Sensor Networks 24
4.3.1 Macro-programming and New Programming Languages 24
4.3.2 Virtual Machines 25

5 Conclusions and Future Work 27
5.1 Conclusions 27
5.2 Future Work 28

6 Other Software and Publications 31
6.1 Software 31
6.2 Publications 32

Bibliography 35

II Papers 43

7 Paper A:
Full TCP/IP for 8-Bit Architectures 45
7.1 Introduction 47
7.2 TCP/IP overview 48
7.3 Related work 50
7.4 RFC-compliance 52
7.5 Memory and buffer management 53
7.6 Application program interface 55
7.7 Protocol implementations 56
7.7.1 Main control loop 56
7.7.2 IP — Internet Protocol 57
7.7.3 ICMP — Internet Control Message Protocol 58
7.7.4 TCP — Transmission Control Protocol 59
7.8 Results 62
7.8.1 Performance limits 62
7.8.2 The impact of delayed acknowledgments 63
7.8.3 Measurements 64
7.8.4 Code size 66
7.9 Future work 69
7.10 Summary and conclusions 70
7.11 Acknowledgments 70
Bibliography 70

8 **Paper B:**
Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors **75**
8.1 Introduction 77
8.1.1 Downloading code at run-time 78
8.1.2 Portability 78
8.1.3 Event-driven systems 79
8.2 Related work 80
8.3 System overview 81
8.4 Kernel architecture 82
8.4.1 Two level scheduling hierarchy 83
8.4.2 Loadable programs 83
8.4.3 Power save mode 84
8.5 Services 85
8.5.1 Service replacement 85
8.6 Libraries 86
8.7 Communication support 87
8.8 Preemptive multi-threading 88
8.9 Discussion 89
8.9.1 Over-the-air programming 89
8.9.2 Code size 89
8.9.3 Preemption 91
8.9.4 Portability 92
8.10 Conclusions 92
Bibliography 93

9 **Paper C:**
Using Protothreads for Sensor Node Programming **97**
9.1 Introduction 99
9.2 Motivation 100
9.3 Protothreads 102
9.3.1 Protothreads versus events 102
9.3.2 Protothreads versus threads 103
9.3.3 Comparison 103
9.3.4 Limitations 105
9.3.5 Implementation 106
9.4 Related Work 107
9.5 Conclusions 108
Bibliography 108

10 Paper D: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems 111
10.1 Introduction . . . 113
10.2 Protothreads . . . 115
10.2.1 Scheduling . . . 116
10.2.2 Protothreads as Blocking Event Handlers . . . 117
10.2.3 Example: Hypothetical MAC Protocol . . . 117
10.2.4 Yielding Protothreads . . . 120
10.2.5 Hierarchical Protothreads . . . 120
10.2.6 Local Continuations . . . 121
10.3 Memory Requirements . . . 122
10.4 Replacing State Machines with Protothreads . . . 123
10.5 Implementation . . . 125
10.5.1 Prototype C Preprocessor Implementations . . . 126
10.5.2 Memory Overhead . . . 130
10.5.3 Limitations of the Prototype Implementations . . . 130
10.5.4 Alternative Approaches . . . 131
10.6 Evaluation . . . 133
10.6.1 Code Complexity Reduction . . . 133
10.6.2 Memory Overhead . . . 140
10.6.3 Run-time Overhead . . . 141
10.7 Discussion . . . 143
10.8 Related Work . . . 143
10.9 Conclusions . . . 145
Bibliography . . . 146

11 Paper E: Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks 151
11.1 Introduction . . . 153
11.2 Scenarios for Software Updates . . . 154
11.2.1 Software Development . . . 154
11.2.2 Sensor Network Testbeds . . . 154
11.2.3 Correction of Software Bugs . . . 155
11.2.4 Application Reconfiguration . . . 155
11.2.5 Dynamic Applications . . . 155
11.2.6 Summary . . . 156

11.3 Code Execution Models and Reprogramming 156
11.3.1 Script Languages 157
11.3.2 Virtual Machines 157
11.3.3 Native Code 157
11.4 Loadable Modules 158
11.4.1 Pre-linked Modules 160
11.4.2 Dynamic Linking 161
11.4.3 Position Independent Code 162
11.5 Implementation 162
11.5.1 The Contiki Operating System 163
11.5.2 The Symbol Table 164
11.5.3 The Dynamic Linker 164
11.5.4 The Java Virtual Machine 167
11.5.5 CVM - the Contiki Virtual Machine 167
11.6 Evaluation 168
11.6.1 Energy Consumption 170
11.6.2 Memory Consumption 176
11.6.3 Execution Overhead 176
11.6.4 Quantitative Comparison 178
11.6.5 Scenario Suitability 181
11.6.6 Portability 183
11.7 Discussion 183
11.8 Related Work 184
11.9 Conclusions 187
Bibliography 187

SICS Dissertation Series **193**

I

Thesis Summary

Chapter 1

Introduction

Twenty years ago the computer revolution put PCs in offices and homes throughout large parts of the western world. Ten years later the Internet revolution connected the computers together in a world-spanning communication network. Today, we stand on the brink of the next revolution: networked embedded systems that connect the physical world together with computers, enabling a large variety of applications such as health and heart resuscitation monitoring [8, 69], wildlife tracking and volcano monitoring [35, 48, 70], building structure monitoring [39], building automation [63], and carbon dioxide monitoring in relation to global warming [41].

It is difficult to estimate the total number of embedded systems in the world today, but it is possible to get a grasp of the magnitude of the area by looking at sales figures for microprocessors. We might expect that PCs account for the bulk of microprocessors because of their widespread use. However, PCs account for only a very small part of the microprocessor market. In 2002, only 2% of all microprocessors sold were used in PCs [66]. The remaining 98% of all microprocessors were sold for use in various types of embedded systems.

Embedded systems typically have much less memory than general-purpose PCs. A normal PC sold in late 2006 had thousands of millions bytes of random access memory (RAM). This is many million times larger than the RAM size in many embedded systems; the microprocessors in many embedded systems have as little as a few hundred or a few thousand bytes of RAM.

It is difficult to estimate the typical or average memory size in embedded systems today, but again we can get a grasp of the magnitude by looking at the microprocessor sales figures. Of the total number of microprocessors sold

in 2002, over 90% were significantly smaller in terms of memory size than a modern PC [66]. In fact, over 50% of all microprocessors were so-called 8-bit processors, which typically only can handle a maximum of 65536 bytes of memory. Because of cost constraints, most of those microprocessors are likely to have considerably less memory than the maximum amount. For example, in 2004 the price of the popular Texas Instruments' MSP430 FE423 microprocessor was nearly 20% higher with 1024 bytes of RAM ($5.95) than the same microprocessor with 256 bytes of RAM ($4.85) [31]. While the price of on-chip memory is likely to decrease in the future, the number of applications for microprocessors will increase. As these applications will require microprocessors with an even lower per-unit cost than today, future microprocessor models are likely to have similar memory configurations as today's microprocessors but at lower per-unit prices.

Most microprocessors in embedded systems are programmed in the C programming language [22]. Their memory constraints make programming them a challenge. Programmers that program general-purpose computers or PCs seldom have to take memory limitations into consideration because of the large amounts of memory available. Moreover, PC microprocessors hardware make techniques such as virtual memory possible, making the memory accessible to the programmer almost limitless. In contrast, the small amounts of memory requires embedded systems programmers to always be aware of memory limitations when programming their systems. Also, most microprocessors for embedded systems do not have the ability to extend the physical memory with virtual memory. In this thesis I use the term *memory constrained* for systems where the programmer explicitly must take memory limitations into consideration when programming the system.

Many embedded systems today communicate with each other. Examples include base stations for mobile telephony, wireless car keys, point of sale terminals, and data logging equipment in trucks. The embedded systems communicate both with other embedded systems and general-purpose computers using a network. In this thesis I call such systems *networked embedded systems*. Networked embedded systems are, just like ordinary embedded systems, often memory constrained.

1.1 Wireless Sensor Networks

A special kind of networked embedded systems are wireless sensor networks. Wireless sensor networks consist of many small wireless networked embedded

systems, equipped with sensors, that form a wireless network through which sensor readings are transmitted [61]. Each sensor node is a networked embedded system. Sensor data is relayed from sensor node to sensor node towards a base station. If a node should break, sensor network routing protocols may reroute the data around the broken node.

A wireless sensor network may consist of up to thousands of sensor nodes. Because of the potential large scale of the sensor network the individual sensors must be small, low cost, and expendable. For this reason, the sensor nodes used in wireless sensor networks typically have memory-constrained microprocessors. Commercially available sensor nodes have between 2 and 10 kilobytes of RAM [1, 59, 62]. Moreover, for sensor networks to run for extended periods of time, the energy consumption of both individual sensor nodes and of the network as a whole is of primary importance. Thus energy consumption is an important performance metric for sensor networks.

1.2 Programming Memory-Constrained Embedded Systems

This thesis is about programming memory-constrained networked embedded systems, such as sensor nodes in wireless sensor networks. I use the word *programming* in the sense of programming-in-the-small [18], not in the sense of programming-in-the-large, and not in the sense of software engineering. Programming-in-the-small is the process of connecting individual program language statements to form program modules, whereas programming-in-the-large is the process of connecting modules into programs or systems. Software engineering is the process of gathering requirements, analyzing the requirements, designing the software system, and implementing it, possibly with a large team of developers. Programming-in-the-small is an important part of the software engineering process as it is what constitutes the final implementation phase. It is also in this phase that memory constraints are most evidently visible to the system developers.

1.3 Research Approach and Method

Throughout the work with this thesis I have taken a pragmatic approach to programming memory-constrained networked embedded systems: I have used the C programming language, the most commonly used programming language

for embedded systems [22], and I have made it a point to make my software work on a wide range of embedded systems platforms. An alternative approach would have been to use uncommon programming languages, develop new programming languages, or develop new hardware platforms. However, this pragmatic approach has enabled me to interact with embedded systems developers working with actual embedded systems and products which would have been very difficult if I had not used the C programming language. Interacting with embedded systems developers has given me insights into many of the actual problems that exist in embedded systems programming and has forced me to build systems that actually work. Moreover, this pragmatic approach also makes the research directly accessible to practitioners. This is one of the reasons behind the large impact of this thesis.

The research method employed in this thesis has been the method of computer systems research [16]: I have built computer systems and conducted experiments with them in order to evaluate a specific method, tool, mechanism, or implementation technique. The systems I have built are software systems: two TCP/IP stacks for memory-constrained systems, lwIP and uIP, and an operating system for memory-constrained systems, Contiki.

My research work has typically gone through two phases, one exploratory phase and one confirmatory phase. In the exploratory phase I have been writing computer programs, either as part of another research project or for personal enjoyment. When programming I have come up with an interesting idea and have become interested in finding out whether the idea is good or not. To test the idea I have formulated an initial hypothesis to verify or falsify. The work has then entered the confirmatory phase. In the confirmatory phase I test the hypothesis that I have developed during the exploratory phase. For the purpose of testing the hypothesis I have built a software system to carry out experiments that either verify or falsify my hypothesis. I have then conducted the experiments and evaluated the results. If the experiments have not supported my hypothesis I have either revised the hypothesis and rebuilt my system, or have abandoned the hypothesis and continued the exploratory phase.

1.4 Research Issues

In this thesis I focus on three topics pertaining to programming memory-constrained networked embedded systems: the use of the TCP/IP protocol suite for memory-constrained embedded systems; a novel programming abstraction that I have named *protothreads*, which is intended to simplify event-driven

programming of memory-constrained systems; and dynamic loading of native code modules in my Contiki operating system for memory-constrained system.

A general theme throughout this research is the use of standard or general-purpose mechanisms. In academic research we are not restricted to standards. Rather, we can freely choose to investigate our own protocols, programming languages, file formats, and mechanisms. This is partly due to that we often work in areas in which no standards have been created. However, even in areas where standards exist or are emerging we often develop and use our own protocols, programming languages, file formats, and mechanisms. Because of this we cannot be sure *why* we choose our own solutions over the standard solutions. Do we go our own way because we want to? Or are we compelled to do it because of the overheads of the standard solutions?

Part of the research in this thesis is about answering the question of how far we can push general-purpose or standard mechanisms before we need to invent our own mechanisms. Or, conversely, how well does general-purpose and standard mechanisms work for memory-constrained systems?

1.4.1 TCP/IP for Memory-Constrained Systems

All systems connected to the global Internet, wireless networks such as WLAN and GPRS, and many local area networks communicate using the standard TCP/IP protocol suite. Due to the prevalence of TCP/IP networks many networked embedded systems are connected to such networks and therefore must be able to communicate using the TCP/IP protocols. However, the TCP/IP protocol suite is often perceived to be "heavy-weight" in that an implementation of the protocols requires large amounts of resources in terms of memory and processing power. This perception can be corroborated by measuring the memory requirements of popular TCP/IP implementations, such as the one in the Linux kernel [28] or in the BSD operating system [52]. The TCP/IP implementations in these systems require many hundreds of kilobytes of RAM and have a code footprint of approximately hundred kilobytes.

The perception that the TCP/IP protocols would require large amounts of memory leads to system designers equipping their embedded systems with large microprocessors. If it was possible to implement the TCP/IP protocol suite in radically less memory, system designers could choose smaller microprocessors for their embedded systems. This would not only make the resulting systems less costly to produce, but would also enable an entire class of smaller embedded systems to communicate using the TCP/IP protocol suite. This is the focus of Paper A.

1.4.2 Protothreads and Event-Driven Programming

Networked embedded systems must handle multiple events that occur concurrently; they must handle both interactions with the physical world and interactions with other systems that communication over the network. To handle concurrency in a small amount of memory, many memory-constrained networked embedded systems are built on an event-driven programming model rather than a multithreaded model [29, 53].

In the multithreaded model concurrency is implemented by designing the system as a set of threads. Each thread is a program that runs concurrently with the other threads in the system. Threads can wait for events to occur. The thread then blocks its execution until it is woken up by the operating system. Each thread requires its own processor stack. In a memory-constrained system, multiple stacks may require a large part of the total memory.

In the event-driven model, the system is not made up of threads but of event handlers that all run on the same stack. Since only one stack is used, the memory requirements are reduced. In the event-driven model all program execution is triggered by internal or external events. When an event occurs, an event handler is invoked by the event dispatcher. Event handlers cannot wait for events to occur but must explicitly return control to the event dispatcher. The fact that event handlers cannot do a blocked wait complicates implementation of high-level logic that cannot be expressed as a single event handler. Such logic must be divided into multiple event handlers where the flow control of the logic is implemented as explicit state machines. Such state machines typically are difficult to write, read, and debug [6, 23, 42, 44, 67].

If it was possible to somehow combine the multithreaded and the event-driven model, perhaps programs could be written in a sequential style without explicit state machines and still have a low memory overhead. Furthermore, it would be advantageous if it was possible to do this in a standard, general-purpose programming language. The study of this problem is initiated in Paper B and continued in Papers C and D.

1.4.3 Dynamic Module Loading

Most operating systems for memory-constrained embedded systems are designed in a monolithic fashion where the operating system, hardware device drivers, and all application programs are compiled into a single monolithic binary that is installed in the read-only memory of the microprocessor in the embedded system. This monolithic approach has its benefits, such as memory

footprint predictability and the opportunity for cross-module optimizations, but makes run-time replacement of applications difficult. The monolithic binary has cross dependencies between the operating system and the applications. It is not possible to replace an application without having to alter the compiled operating system code. To replace an application, the entire monolithic binary must be replaced with a new monolithic binary that includes the new application and an updated version of the compiled operating system code. When the new binary has been installed in read-only memory, the system must be rebooted.

Radio communication is the primary energy consumer in many wireless sensor network hardware platforms. By reducing the amount of data that needs to be transmitted across the network when performing a software update, the energy consumption is reduced. Early methods for updating software in wireless sensor networks required the entire monolithic binary to be transmitted over the network [33]. Even if only a single application needs to be updated, the entire operating system binary must be transfered, incurring a large energy overhead.

If it was possible to transmit only the parts of the system that needed to be updated the energy consumption of software updates would be lower. Furthermore, it would be advantageous if it was possible to do this by using standard or general-purpose mechanisms. This is investigated in Papers B and E.

1.5 Thesis Structure

This thesis consists of a thesis summary followed by five papers that are published in peer-reviewed conference and workshop proceedings. Conference proceedings are the primary venue for scientific publishing in the area of this thesis.

The rest of this thesis is structured as follows. Chapter 2 summarizes the scientific contributions of this thesis and reviews the impact that the research in the thesis has. Chapter 3 summarizes the papers of the thesis and their individual contributions. Related work is reviewed in Chapter 4. I conclude the thesis and present ideas for future work in Chapter 5. Chapter 6 presents relevant software I have developed but that is not strictly part of the thesis and a list of papers authored or co-authored by me during the thesis work but not included in this thesis. Chapters 7, 8, 9, 10, and 11 contain the five papers of the thesis.

Chapter 2

Scientific Contributions and Impact

2.1 Scientific Contributions

In this thesis I make three main scientific contributions. My first contribution is showing that the TCP/IP protocols can be implemented efficiently enough to run even in memory-constrained embedded systems, but that it leads to significantly lower network throughput. My second contribution is protothreads, an intentionally simple yet very powerful programming abstraction. I show that protothreads significantly reduce the complexity of event-driven programs for memory-constrained systems. My third contribution is that I show that dynamic linking and loading of native code modules is feasible for memory-constrained operating systems. I quantify the energy consumption and evaluate dynamic loading, linking, and execution of native code against loading and execution of virtual machine code.

Prior to the work presented in this thesis, the general consensus was that the TCP/IP protocol suite was inherently unsuited for memory-constrained systems because an RFC standards-compliant implementation of the TCP/IP protocols would require far too much resources in terms of memory to even be possible to fit in a memory-constrained embedded system. In this thesis I show that RFC-compliant TCP/IP can be used in systems an order of magnitude smaller than previously believed possible.

My protothreads programming abstraction shows that it is possible to do se-

quential programming on event-driven systems without the memory overhead of full multi-threading. Protothreads are an intentionally simple but powerful mechanism that significantly reduces the complexity of state machine-based event-driven programs for memory-constrained systems. Out of seven state machine-based event driven programs rewritten with protothreads, the state machines could be almost completely removed for all of them and the number of lines of code was reduced with 31% on average, at the price of a few hundred bytes increase in size of the compiled code.

The memory overhead of protothreads is small. A program written with protothreads uses only one byte more memory than the equivalent program implemented as a state machine-based event-driven program. The run-time overhead of a protothread is small; the run-time of a protothread is only a few processor cycles higher than that of the equivalent state machine-based event-driven program. This makes protothreads usable even in time-critical code such as hardware interrupt handlers.

My Contiki operating system is the first operating system in the wireless sensor network community to support loadable modules. Previous operating systems for sensor networks require the entire system and applications to be compiled into a single monolithic binary that is loaded onto each sensor node. In such systems the entire system binary must be replaced when reprogramming a single module. With loadable modules, only the module that is changed needs to be replaced. Contiki is also the first operating system for sensor networks to support both event-driven and threaded programming.

In this thesis I quantify the energy cost of run-time dynamic loading and linking of native code modules in the de facto standard ELF object code format. My results show that the overhead of run-time dynamic linking is fairly small and that the overhead mainly is due to the ELF file format and not due to the dynamic linking mechanism as such. Furthermore, when comparing dynamic linking and loading of native code modules with virtual machine modules the results suggest that combinations of virtual machine code and dynamically loaded native code provide better energy efficiency than pure native code modules and pure virtual machine code modules.

2.2 Impact

The impact of the research in this thesis has been, and continues to be, large. The lwIP and uIP software developed as part of this thesis has been adopted by well over one hundred companies world-wide in a wide variety of embed-

ded devices. Examples include satellite systems, oil boring and oil pipeline equipment, TV transmission devices, equipment for color post-processing of movies, world-wide container monitoring and security systems, switches and routers, network cameras, and BMW racing car engines. The software is also used in system development kits from hardware manufacturers including Analog Devices, Altera, and Xilinx, which greatly increases the dissemination of the software. Articles in professional embedded systems developer magazines have been written, by others, on porting the uIP software for new hardware platforms [9]. The software is also covered in books on embedded systems and networking [34, 50] and is included in embedded operating systems [2, 14, 65]. The lwIP and uIP homepages have for a few years been among the top five hits for Google searches such as TCP/IP stack and embedded TCP/IP.

The Contiki operating system has become a well-known operating system in the wireless sensor network community and is used by several research projects in the area. Many ideas from Contiki such as dynamic module loading and the optional multi-threading library have also been adopted by other operating systems for wireless sensor networks [26, 51]. The dynamic loader mechanism in Contiki has also been investigated for use by Ericsson Mobile Platforms as part of the hardware platform used in many of today's 3G mobile telephones.

Protothreads are currently used by numerous embedded software developers and have been recommended twice in acclaimed embedded developer Jack Ganssle's Embedded Muse newsletter [20]. Protothreads have also been ported, by others, to other programming languages and operating systems [40, 58].

The papers and the software in this thesis are used in advanced courses on embedded systems and sensor networks at many universities and institutions throughout the world. The papers are cited by many scientific papers in the area of wireless sensor networks. In early 2007 the Google Scholar citation index [3] reports a total of 121 citations of the first three papers in this thesis. The last two papers had not yet been indexed by Google Scholar.

Chapter 3

Summary of the Papers and Their Contributions

The thesis is a collection of five papers, Paper A, B, C, D, and E. All papers are published in peer-reviewed conference and workshop proceedings. Conference proceedings are the primary venue for scientific publishing in the area of this thesis. I have presented all papers at the conferences and workshops at which they appeared.

Papers A, D, and E are published at top-tier conferences, ACM MobiSys 2003 and ACM SenSys 2006. ACM MobiSys is a high-quality single track conference. ACM SenSys is regarded as the most prestigious conference in the area of wireless sensor networks. Paper B was published in the first instance of a now established high quality workshop, IEEE EmNets 2004. Paper C was published at the REALWSN 2005 workshop on real-world wireless sensor networks.

Paper A presents and evaluates the lwIP and uIP TCP/IP stacks. The event-driven nature of uIP forms the basis of the Contiki operating system introduced in Paper B. Paper B presents Contiki and the dynamic module loading mechanism and the per-process optional multi-threading in Contiki. The dynamic loading mechanism in Contiki is further developed and evaluated in Paper E. The multi-threading mechanism in Contiki, presented in Paper B, is the first step towards the protothreads mechanism that I introduce in Paper C. Paper D refines, extends, and evaluates the protothreads mechanism. Paper C also includes a qualitative comparison between protothreads, events, and threads that is not included in Paper D. Papers B, C, D, and E show how the research

has progressed from development of the underlying mechanisms in Contiki in Paper B and the novel protothreads mechanism in Paper C, to more refined mechanisms and their evaluations in Paper D and E.

3.1 Paper A: Full TCP/IP for 8-Bit Architectures

Adam Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (ACM MobiSys 2003)*, San Francisco, USA, May 2003.

Summary. The TCP/IP protocol suite is the family of protocols used for communication over the global Internet, and is often used in private networks such as local-area networks and corporate intranets. In order to attach a device to the network, the device must be able to use the TCP/IP protocols for communication.

This paper presents two small implementations of the TCP/IP protocol stack with slightly different designs; lwIP, which is designed in a modular and generic fashion, similar to how large-scale protocol implementations are designed, and uIP which is designed in a minimalistic fashion and only contains the absolute minimum set of features required to fulfill the protocol standards. In order to reduce the code size and the memory requirements, the uIP implementation uses an event-based API which is fundamentally different from the most common TCP/IP API, the BSD socket API.

As expected, measurements from an actual system running the implementations show that the smaller uIP implementation provides a very low throughput, particularly when sending data to a PC host. However, small systems running uIP usually do not produce enough data for the performance degradation to become a serious problem.

Contribution. The main contribution of this paper is that it refutes the common conception that the TCP/IP protocol suite is too “heavy-weight” to be possible to fully implement on tiny devices. At the time this paper was written, most TCP/IP protocol stack implementations were designed for workstations and server-class systems, where communication performance was the primary concern. This caused a wide-spread belief that tiny devices such as sensor network nodes would be too constrained to be able to fully implement the TCP/IP stack. There were also a number of commercial implementations of the TCP/IP stack intended for embedded devices, where the protocols in the TCP/IP suite had been modified in order to reduce the code size and memory usage of their implementation. Such implementations are problematic as they may cause in-

teroperability problems with other TCP/IP implementations. This paper shows that it is possible to implement the TCP/IP stack in a very small code size and with a very small memory usage, without modifying the protocols. However, a small TCP/IP stack will not be able to achieve as high throughput as a larger implementation.

3.2 Paper B: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors

Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE Emnets 2004)*, Tampa, Florida, USA, November 2004.

Summary. The ability to reprogram wireless sensor networks using the radio reduces the time for the software development cycle. However, operating systems for sensor networks require the full system image to be replaced when reprogramming a sensor network over the radio. This paper presents Contiki, a lightweight and flexible operating system for tiny networked embedded devices. Unlike other operating systems for sensor networks, Contiki has the ability to selectively load and unload individual programs. This reduces the amount of data that needs to be transmitted when reprogramming a sensor network over the radio, since the entire system binary does not need to be transfered. We show that the dynamic loading can be done while keeping the memory footprint down.

Contiki supports two kinds of concurrency mechanisms: an event-driven interface and a preemptive multi-threading interface. The advantages of an event-driven model is that it is possible to implement using very small amounts of memory. Preemptive multi-threading, on the other hand, requires comparatively large amounts of memory to hold per-thread stacks. Furthermore, there are types of programs that are unsuited for the event-driven model but work well with preemptive multi-threading. Computationally intensive programs such as encryption algorithms are typical examples of this.

Unlike other operating systems, Contiki leverages both models by basing the system on an event-driven kernel and implementing preemptive multi-threading as an optional application library. This allows using preemptive multi-threading on a per-program basis. Experiments show that a Contiki sys-

tem is able to continue to respond to events in a timely manner while performing a long-running computation as a preemptible thread.

Contribution. The main contribution of this paper is introducing the idea of loadable modules for a sensor network operating system. Furthermore, the combination of multi-threaded and event-driven operation in Contiki was not previously used in sensor network operating systems.

Per-author contributions. I developed the Contiki system and I wrote the paper. Contiki was developed and the paper was written on my initiative. The idea of providing preemptive multi-threading as an application library on top of the event-driven Contiki kernel was formed in cooperation with Björn Grönvall who also took part in writing the paper. All other ideas in the paper are mine. Thiemo Voigt was involved in the writing of the paper.

3.3 Paper C: Using Protothreads for Sensor Node Programming

Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. Using protothreads for sensor node programming. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN 2005)*, Stockholm, Sweden, June 2005.

Summary. Memory-constrained systems are often programmed using the event-driven model rather than a multi-threaded model because multi-threading has a higher memory overhead than the event-driven model. The problem with the event-driven model is that sequencing high-level operations must be done in a state machine-style. State machine-programs written in C are difficult to understand, debug, and maintain.

Protothreads is a lightweight mechanism developed to make it possible to write high-level sequences as a sequence of source code statements rather than as explicit state machines.

Contribution. This paper introduces the idea of protothreads as a lightweight mechanism to provide sequential programming on top of an event-driven system. The protothreads mechanism is intentionally simple; it requires only two bytes of memory per protothread and it is possible to implement protothreads using only 6 lines of C code.

Per-author contributions. I invented protothreads, came up with their implementation in C, and wrote the paper. The ideas in the paper are mine and the work was done and the paper was written on my initiative. Oliver Schmidt and I discussed and further developed the ways protothreads could be implemented. Thiemo Voigt was involved in the writing of the paper.

3.4 Paper D: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (ACM SenSys 2006)*, Boulder, Colorado, USA, November 2006.

Summary. In this paper we significantly extend the results of Paper C in that we measure the usefulness of protothreads by reimplementing a set of state machine-based programs with protothreads. We show that protothreads can significantly reduce the complexity in terms of number of lines of code for the rewritten programs. Furthermore, we show that the execution time overhead of protothreads is small: only a few processor cycles per invocation. This is small enough to make protothreads usable even in time-critical code such as interrupt handlers. However, protothreads incur a code size overhead of a few hundred bytes per program.

Contribution. The contribution of this paper is that we show that protothreads, a very simple mechanism, significantly reduce the complexity of state machine-based event-driven programs without increasing memory requirements or execution time.

Per-author contributions. I invented protothreads, came up with their implementation in C, designed and carried out the evaluation, and wrote the paper. The ideas in the paper are mine and work was done and the paper was written on my initiative. Oliver Schmidt and I discussed and further developed some of the ways protothreads could be implemented. Thiemo Voigt was involved in the writing of the paper. Muneeb Ali wrote two paragraphs in the Related Work section. Measuring the code size of the rewritten programs was suggested by SenSys paper shepherd Philip Levis.

3.5 Paper E: Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks

Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*

(ACM SenSys 2006), Boulder, Colorado, USA, November 2006.

Summary. Reprogramming wireless sensor networks is a very useful feature in a large number of situations. Many different methods for reprogramming wireless sensor networks have been investigated. In this paper we introduce a new reprogramming method: dynamic linking of native code using the standard ELF object code file format.

We compare the energy efficiency of dynamic linking with three other reprogramming mechanisms: full image replacement, an application-specific virtual machine, and a standard Java virtual machine. We conclude that the ELF format has a high overhead but that the overhead is due to the file format and not due to the dynamic linking mechanism. Furthermore, we show that, from an energy consumption perspective, combinations of virtual machine code and native code may prove to be the most energy efficient alternative.

Contribution. This paper shows for the first time that a standard mechanism for loading code, dynamic linking of ELF object code files, is doable and feasible even in resource-constrained wireless sensor network nodes. Furthermore, the quantification of the energy cost of dynamic linking and the quantification of the energy costs of native code versus virtual machine code and Java code for sensor networks is a contribution on its own.

Per-author contributions. I designed and implemented the Contiki ELF loader, the Contiki virtual machine and its compiler, did the initial porting of the leJOS Java virtual machine, designed and carried out the experimental evaluation, and wrote the paper. The ideas in the paper are mine and the work was done and the paper was written on my initiative. Niclas Finne and Joakim Eriksson did the largest part of the porting of the leJOS Java virtual machine to Contiki. Niclas also took part in performing the energy measurements in the paper and Joakim wrote the initial version of the subsection on the Java virtual machine. Thiemo Voigt was involved in the writing of the paper.

Chapter 4

Related Work

4.1 Small TCP/IP Implementations

There are several small TCP/IP implementations for memory-constrained embedded systems. However, most of those implementations refrain from implementing required protocol mechanisms in order to reduce the complexity of the implementation. The resulting implementation may therefore not be fully compatible with other TCP/IP implementations. Hence communication between the memory-constrained system and another system may not be possible.

Many small TCP/IP implementations are tailored for a specific application, such as running a web server. Tailoring the TCP/IP implementation for a specific application makes it possible to significantly reduce the implementation complexity. However, such an implementation does not provide a general communications mechanism that can be used for other applications. The PICmicro stack [11] is an example of such a TCP/IP implementation. Unlike such implementations, the uIP and lwIP implementations are not designed for a specific application.

Other implementations rely on the assumption that the small embedded device will always be communicating with a full-scale TCP/IP implementation running on a PC or work-station class device. Under this assumption it is possible to remove certain mechanisms that are required for full compatibility. Specifically, support for IP fragment reassembly and for TCP segment size variation are two mechanisms that are required by the standard but are often left out. Examples of such implementations are Texas Instrument's MSP430 TCP/IP stack [17] and the TinyTCP code [15]. Unlike those implementations,

the uIP or the lwIP stack support both IP fragment reassembly and a variable maximum TCP segment size.

In addition to the TCP/IP implementation for small embedded systems, there is a large class of TCP/IP implementations for embedded systems with less constraining limitations. Typically, such implementations are based on the TCP/IP implementation from the BSD operating system [52]. These implementations do not suffer from the same problems as the tailored implementations. However, such implementations require too large amounts of resources to be feasible for memory-constrained embedded systems. Such implementations are typically orders of magnitude larger than uIP.

4.2 Operating Systems for Wireless Sensor Networks

TinyOS [29] is probably the earliest operating system that directly targets the specific applications and limitations of sensor devices. TinyOS is built around a lightweight event scheduler where all program execution is performed in tasks that run to completion. TinyOS uses a special description language for composing a system of smaller components [21] that are statically linked with the kernel to a complete monolithic binary image of the system. After linking, modifying the system is not possible [42]. The Contiki system is also designed around a lightweight event-scheduler, but is designed to allow loading, unloading, and replacing modules at run-time.

To load new applications in an operating system where the entire system is compiled into a monolithic binary, the entire system binary must be updated. Since the energy consumption of distributing code in sensor networks increases with the size of the code to be distributed several attempts have been made to reduce the size of the code to be distributed. Reijers and Langendoen [60] produce an edit script based on the difference between the modified and original binary system image. After a number of optimizations including architecture-dependent ones, the script is distributed throughout the network. A similar approach has been developed by Jeong and Culler [32] who use the rsync algorithm to generate the difference between modified and original executable. Koshy and Pandey's diff-based approach [37] reduces the amount of flash rewriting by modifying the linking procedure so that functions that are not changed are not shifted.

FlexCup [49] uses another approach to enable run-time installation of TinyOS software components: loadable applications include the symbolic

names of functions in the loaded application. When an application is loaded, the symbolic names are resolved and the loaded binary is updated. FlexCup uses a complete duplicate image of the system's monolithic binary image that is stored in external flash ROM. The copy of the system image is used for constructing a new system image when a new program has been loaded. However, FlexCup uses a non-standard format and is less portable than the loader in Contiki. Further, FlexCup requires a reboot after a program has been installed, requiring an external mechanism to save and restore the state of all other applications as well as the state of running network protocols across the reboot. Contiki does not need to be rebooted after a program has been installed. Also the Contiki dynamic linker does not alter the core image when programs are loaded and therefore no external copy of the core image needs to be stored.

In order to provide run-time reprogramming for TinyOS, Levis and Culler have developed Maté [42], a virtual machine for TinyOS devices. Code for the virtual machine can be downloaded into the system at run-time. The virtual machine is specifically designed for the needs of typical sensor network applications. Similarly, the MagnetOS [10] system uses a virtual Java machine to distribute applications across the sensor network. The advantages of using a virtual machine instead of native machine code is that the virtual machine code can be made smaller, thus reducing the energy consumption of transporting the code over the network. One of the drawbacks is the increased energy spent in interpreting the code—for long running programs the energy saved during the transport of the binary code is instead spent in the overhead of executing the code. Contiki does not suffer from the executional overhead since modules loaded into Contiki are compiled to native machine code.

The Mantis system [5] uses a traditional preemptive multi-threaded model of operation. Mantis enables reprogramming of both the entire operating system and parts of the program memory by downloading a program image onto EEPROM, from where it can be burned into flash ROM. Due to the multi-threaded semantics, every Mantis program must have stack space allocated from the system heap, and locking mechanisms must be used to achieve mutual exclusion of shared variables. In Contiki, only programs that explicitly require multi-threading need to allocate an extra stack.

Systems that offer loadable modules besides Contiki include SOS [26] and Impala [46]. Impala features an application updater that enables software updates to be performed by linking in updated modules. Updates in Impala are coarse-grained since cross-references between different modules are not possible. Also, the software updater in Impala was only implemented for much more resource-rich hardware than the memory-constrained systems considered

in this thesis. SOS [26], which was published after Contiki, is similar in design to Contiki: SOS consists of a small kernel and dynamically-loaded modules. However, unlike Contiki SOS uses position independent code to achieve relocation and jump tables for application programs to access the operating system kernel. Application programs can register function pointers with the operating system for performing inter-process communication. Position independent code is not available for all platforms, however, which limits the applicability of this approach.

4.3 Programming Models for Wireless Sensor Networks

While I use the C programming language throughout this thesis, others have investigated the use of new programming languages and programming models for programming memory-constrained networked embedded systems.

4.3.1 Macro-programming and New Programming Languages

Research in the area of software development for sensor networks has led to a number of new abstractions that aim at simplifying the programming of sensor networks [4, 13]. Approaches with the same goal include macro-programming of aggregates of sensor nodes [19, 24, 56, 68], high-level database abstractions of sensor networks [47], and network neighborhood abstractions [54, 68, 71]. The work in this thesis differs from these sensor network programming abstractions in that I target the difficulty of low-level event-driven programming of individual sensor nodes, programming-in-the-small, rather than the difficulty of developing application software for sensor networks, programming-in-the-large.

Kasten and Römer [36] have also identified the need for new abstractions for managing the complexity of event-triggered state machine programming. They introduce OSM, a state machine programming model based on Harel's StateCharts[27] and use the Esterel language. The model reduces both the complexity of the implementations and the memory usage. Their work is different from protothreads in that they help programmers manage explicit state machines, whereas protothreads are designed to reduce the number of explicit state machines. Furthermore, OSM requires support from an external OSM

compiler to produce the resulting C code, whereas the prototype implementations of protothreads only make use of the regular C preprocessor.

Script languages are another alternative for programming embedded systems. Script languages do not need special compilers as the script code is interpreted by the embedded systems. However, existing structured script languages are typically too large for the memory-constraints considered in this thesis. The SensorWare system [12] uses a reduced version of Tcl to provide a script-based programming environment for sensor networks. However, the system is designed for sensor nodes with an order of magnitude more memory resources than the systems considered in this thesis; for example, the SensorWare system occupies 180 kilobytes of memory.

Rappit [25] is a development framework for scripting languages for embedded systems. Rappit uses a host environment running Python that sends commands to the embedded systems. The host environment runs on a resource-rich server system outside of the sensor network that translates commands into simpler messages that are executed by the embedded system. Tapper [72] is a command language and lightweight stack-based script engine for sensor nodes built with Rappit. The Tapper language provides primitives for accessing hardware devices such as analog to digital converters and for sending and receiving radio packets. Both Rappit and Tapper require assistance from a host system for interpreting the commands.

4.3.2 Virtual Machines

Virtual machines have been investigated for sensor networks as an approach to reduce the distribution energy costs for software updates. The code size of the programs running on top of the virtual machine can be kept to a minimum since the virtual machine can be tailored to the needs of an application for a specific domain such as sensor networks. The drawback of virtual machines is the increased execution overhead over native code. The virtual machine approach is different from the approach taken in this thesis as virtual machines need both special compilers that can produce code for the virtual machine as well as a virtual machine running on the embedded system.

Maté [42] was the first virtual machine specifically targeted to wireless sensor networks. Maté is a stack-based virtual machine that runs on top of TinyOS. Maté instructions are 8 bits wide. Each virtual machine instruction is executed in a separate TinyOS run-to-completion task. Levis et al. [43] have further investigated the use of application specific virtual machines (ASVMs) that are compiled for the needs of a particular application or set of applications. Other

examples of stack-based virtual machines for sensor networks are DVM [7] and my CVM Contiki virtual machine (Paper E). There are also Java-based virtual machines for sensor networks apart from the Java VM in Paper E, such as the VM* system [38].

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis I investigate three aspects of programming memory-constrained networked embedded systems: the use of the TCP/IP protocol stack for memory-constrained systems; the novel protothread programming abstraction for state machine-based programs for event-driven systems; and the use of dynamic loading and linking of native code in an operating system for memory-constrained embedded systems. A general theme throughout this work is how applicable standard or general-purpose mechanisms and methods are to memory-constrained systems. I have identified and quantified trade-offs in both the use of general-purpose mechanisms for memory-constrained systems and the use of a general-purpose programming language for reducing the complexity of memory-efficient programming for memory-constrained systems.

In this thesis I show that the standard TCP/IP protocol stack can be implemented efficiently enough to be usable even in memory-constrained systems, but that such an implementation leads to a significant decrease in network throughput. Furthermore, the results show that protothreads reduces the complexity of state machine-based event-driven programs, while having a very small memory and run-time overhead. Finally, the Contiki operating system shows that dynamic loading and run-time dynamic linking of native code is a feasible mechanism for memory-constrained networked embedded systems.

There are at least two conclusions that can be drawn from my research. First, in many cases it is possible to use standard protocols and mechanisms developed for general-purpose computers even in memory-constrained embedded systems. However, there are trade-offs in terms of both memory footprint and energy. Second, protothreads show that it is possible to combine features from multi-threading and event-driven programming; sequential programming from the multi-threaded model and a small memory overhead from the event-driven model. Since a slightly limited version of protothreads can be implemented in the general-purpose C programming language, it is possible to do memory-efficient sequential programming in C without requiring the use of a special-purpose programming language. However, there are trade-off due to the limitations of the C-based implementation of protothreads.

5.2 Future Work

The research in this thesis can be continued in two ways: extensions to or continuations of the research presented in this thesis or application of the methods and approaches used in this research to new problems or problem domains.

There is one immediate extension to the work in this thesis; the current implementation of protothreads does not implement the full protothreads model because of the limitations of the C preprocessor. By implementing a special pre-compiler that would translate protothread-based code into regular C code it is possible to implement the full model. This also makes it possible to minimize the memory overhead of protothreads.

The work on energy-efficient loading of program modules can be continued in at least two ways. Firstly, the energy consumption for transferring loadable modules in Contiki can be reduced by compressing the modules with loss-less compression techniques before the modules are transmitted across the network. Investigating the use of standard compression techniques and the trade-offs in terms of computational energy consumption is an interesting continuation of the work on the dynamic loader in Contiki. Secondly, it would be interesting to compare the use of dynamic module loading in Contiki with differential-based mechanisms for loading modules in monolithic operating system. Investigating the trade-offs and how they vary with different applications and network types would be interesting.

Finally, the approaches and methods used in this thesis can be applied to nearby problem areas. The memory management scheme used in uIP, where a single buffer is used for both incoming and outgoing packets, might be possible

to apply to other types of network protocols, such as power cycling MAC protocols for sensor networks. By using a single buffer, rather than a multi-buffer management scheme, it might be possible to reduce both the data memory requirements and the code footprint. However, additional mechanisms might be needed to ensure that no packet queuing is required.

The research in this thesis suggests that there is a relation between implementation complexity in terms of memory footprint and the achievable application performance; the smaller uIP implementation achieves much lower network throughput than the larger lwIP implementation. A system designer that requires high throughput can choose the larger lwIP stack, whereas a system designer that requires a low memory footprint, but does not have any throughput requirements, can choose the smaller uIP stack. It would be interesting to investigate if this relation exists in other types of protocols as well. For example, many protocols for distributing software updates through sensor networks have been developed [30, 33, 45, 55, 64], each having different properties in terms of both performance and footprint. A system designer would want to be able to choose how much system resources to commit to the software update feature and what kind of performance to expect from the feature, depending on the requirements of particular deployments. By implementing a set of software dissemination protocols and measuring their memory footprint and achievable throughput it might be possible to quantify trade-offs between memory footprint and performance.

The systems in this thesis are built using a "bottom-up" approach rather than a "top-down" approach. Instead of breaking down the system into smaller components that are independently implemented and composed into a complete system, I have started from the bottom with a set of simple building blocks from which I have built a system that is similar to, but not always equivalent to, a system built from the top down. It would be interesting to use this approach for building new programming mechanisms such as abstractions for programming of aggregates of memory-constrained networked embedded systems, so-called macro-programming. Other approaches towards building macro-programming systems have started by defining the programming interface and decomposed this into smaller modules that are executed by each individual node, e.g. the work by Newton et al. [56, 57] and Gummadi et al. [24]. By instead building upwards from a set of small and simple node-level abstractions to an abstract interface for network-level programming it may be possible to build macro-programming systems that are more efficient in terms of both implementation complexity and network communication than systems built in a top-down fashion.

Chapter 6

Other Software and Publications

6.1 Software

In addition to the software I have developed as part of this thesis I have written a number of other programs that are relevant to the research in this thesis.

Miniweb A proof-of-concept implementation of selected parts of the TCP/IP protocol stack together with a simple web server. The code consists of 400 lines of C and uses only 30 bytes of RAM. Although the implementation sacrifices a large number of protocol mechanisms, it does include TCP's congestion control mechanisms. The code is not intended for actual use. The source code is available at http://www.sics.se/~adam/miniweb/.

phpstack A proof-of-concept implementation of an application-specific TCP/IP stack together with a web server written in 600 lines of PHP code. The TCP/IP stack is more simplified than the Miniweb stack and can only participate in single-segment sessions. Thus pages served by the web server are limited to 1460 bytes. If the program had been implemented in a low-level language such as C, it would only have required a handful bytes of RAM. The code is not intended for actual use. The source code is available at http://www.sics.se/~adam/phpstack/.

VNC client, SMTP client, HTTP client, IRC client, and web browser
Proof-of-concept implementations of a number of application-layer protocols built on top of uIP and Contiki. The purpose was to investigate if the event-driven interface of uIP was expressive enough to implement complex application layer protocols. However, it turned out that many of the protocols required complex state machines on top of the event-driven interface of uIP. This insight lead me to develop protothreads. The source code is available at http://www.sics.se/contiki/.

6.2 Publications

The following peer-reviewed publications were authored or co-authored by me during the work with this thesis but are not included in the thesis.

- Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, USA, November 2006.

- Helena Rivas, Thiemo Voigt, and Adam Dunkels. A simple and efficient method to mitigate the hot spot problem in wireless sensor networks. In *Workshop on Performance Control in Wireless Sensor Networks*, Coimbra, Portugal, May 2006.

- Muneeb Ali, Umar Saif, Adam Dunkels, Thiemo Voigt, Kay Römer, Koen Langendoen, Joseph Polastre, and Zartash Afzal Uzmi. Medium access control issues in sensor networks. *ACM SIGCOMM Computer Communication Review*, April 2006.

- Thiemo Voigt, Adam Dunkels, and Torsten Braun. On-demand construction of non-interfering multiple paths in wireless sensor networks. In *Proceedings of the 2nd Workshop on Sensor Networks at Informatik 2005*, Bonn, Germany, September 2005.

- Adam Dunkels, Richard Gold, Sergio Angel Marti, Arnold Pears, and Mats Uddenfeldt. Janus: An architecture for flexible access to sensor networks. In *First International ACM Workshop on Dynamic Interconnection of Networks (DIN'05)*, Cologne, Germany, September 2005.

- Thiemo Voigt and Adam Dunkels. The impact of knowledge about neighbors on the efficiency of geographic routing. In *Proceedings of Radio Sciences and Communication RVK'05*, June 2005.

- Torsten Braun, Thiemo Voigt, and Adam Dunkels. Energy-efficient TCP operation in wireless sensor networks. *PIK Journal Special Issue on Sensor Networks*, 2005.

- Hartmut Ritter, Jochen Schiller, Thiemo Voigt, Adam Dunkels, and Juan Alonso. Experimental Evaluation of Lifetime Bounds for Wireless Sensor Networks. In *Proceedings of the Second European Workshop on Sensor Networks (EWSN2005)*, Istanbul, Turkey, January 2005.

- Adam Dunkels, Thiemo Voigt, Niclas Bergman, and Mats Jönsson. The Design and Implementation of an IP-based Sensor Network for Intrusion Monitoring. In *Swedish National Computer Networking Workshop*, Karlstad, Sweden, November 2004.

- Thiemo Voigt, Adam Dunkels, and Juan Alonso. Reliability in distributed TCP caching. In *Workshop on Sensor Networks Workshop at Informatik 2004*, Ulm, Germany, September 2004.

- Adam Dunkels, Thiemo Voigt, Juan Alonso, and Hartmut Ritter. Distributed TCP caching for wireless sensor networks. In *Proceedings of the Third Annual Mediterranean Ad Hoc Networking Workshop (MedHocNet 2004)*, June 2004.

- Thiemo Voigt, Hartmut Ritter, Jochen Schiller, Adam Dunkels, and Juan Alonso. Solar-aware Clustering in Wireless Sensor Networks. In *Proceedings of the Ninth IEEE Symposium on Computers and Communications*, June 2004.

- Juan Alonso, Adam Dunkels, and Thiemo Voigt. Bounds on the energy consumption of routings in wireless sensor networks. In *Proceedings of the 2nd WiOpt, Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, Cambridge, UK, March 2004.

- Adam Dunkels, Thiemo Voigt, and Juan Alonso. Making TCP/IP Viable for Wireless Sensor Networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session*, Berlin, Germany, January 2004.

- Adam Dunkels, Thiemo Voigt, Juan Alonso, Hartmut Ritter, and Jochen Schiller. Connecting Wireless Sensornets with TCP/IP Networks. In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, Frankfurt (Oder), Germany, February 2004.
- Laura Marie Feeney, Bengt Ahlgren, Assar Westerlund, and Adam Dunkels. Spontnet: Experiences in configuring and securing small ad hoc networks. In *Proceedings of The Fifth International Workshop on Network Appliances (IWNA5)*, Liverpool, UK, October 2002.

Bibliography

[1] Crossbow mica motes. Web page.
URL: http://www.xbow.com/

[2] eCos Embedded Configurable Operating System. Web page.
URL: http://sources.redhat.com/ecos/

[3] Google Scholar. Web page.
URL: http://scholar.google.com/

[4] T. Abdelzaher, J. Stankovic, S. Son, B. Blum, T. He, A. Wood, and C. Lu. A communication architecture and programming abstractions for real-time embedded sensor networks. In *Workshop on Data Distribution for Real-Time Systems*, Providence, RI, USA, May 2003.

[5] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 50–59, San Diego, CA, USA, September 2003.

[6] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, pages 289–302, 2002.

[7] R. Balani, S. Han, R. Rengaswamy, I. Tsigkogiannis, and M. B Srivastava. Multi-level software reconfiguration for sensor networks. In *Proceedings of the 6th ACM/IEEE Conference on Embedded Software (EMSOFT '06)*, Seoul, Korea, October 2006.

[8] H. Baldus, K. Klabunde, and G. Muesch. Reliable set-up of medical body-sensor networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN'04)*, Berlin, Germany, January 2004.

[9] D. Barnett and A. J. Massa. Inside the uIP Stack. *Dr. Dobb's Journal*, February 2005.

[10] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Opererating Systems Review*, 36(2):1–5, 2002.

[11] J. Bentham. *TCP/IP Lean: Web servers for embedded systems*. CMP Books, October 2000.

[12] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, San Francisco, USA, May 2003.

[13] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A programming model for event-driven embedded systems. In *Proc. of the 18th Annual ACM Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA, March 2003.

[14] A. Christian and J. Healey. Gathering motion data using featherweight sensors and TCP/IP over 802.15.4. In *Proceedings of the IEEE International Symposium on Wearable Computers, On-Body Sensing Workshop*, Osaka, Japan, October 2005.

[15] G. H. Cooper. TinyTCP. Web page. 2002-10-14.
URL: http://www.csonline.net/bpaddock/tinytcp/

[16] National Research Council. *Academic Careers for Experimental Computer Scientists and Engineers*. National Academy Press, 1994.

[17] A. Dannenberg. MSP430 internet connectivity. SLAA 137, November 2001. Avalible from www.ti.com.

[18] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, Los Angeles, California, 1975.

[19] C. Frank and K. Römer. Algorithms for generic role assignment in wireless sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems (SenSys '05)*, San Diego, California, USA, November 2005.

[20] J. Ganssle. The embedded muse. Monthly newsletter.
URL: http://www.ganssle.com/tem-back.htm

[21] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, San Diego, California, USA, June 2003.

[22] J. Grenning. Why are you still using C? *Embedded Systems Design*, April 2003.

[23] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. Programming for pervasive computing environments. *ACM Transactions on Computer Systems*, January 2002.

[24] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *Proc. of Distributed Computing in Sensor Systems (DCOSS)'05*, Marina del Rey, CA, USA, June 2005.

[25] J. Hahn, Q. Xie, and P. H. Chou. Rappit: framework for synthesis of host-assisted scripting engines for adaptive embedded systems. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '05)*, Jersey City, NJ, USA, September 2005.

[26] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys '05)*, Seattle, WA, USA, June 2005.

[27] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[28] T. F. Herbert. *The Linux TCP/IP Stack: Networking For Embedded Systems*. Charles River Media, 2004.

[29] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusets, USA, November 2000.

[30] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys '04)*, Baltimore, Maryland, USA, November 2004.

[31] Texas Instruments. Analog enewsletter. Web page, May 2004. Visited 2006-12-21.
URL: http://focus.ti.com/en/download/aap/enewsletters/0405enewsletter.htm

[32] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON (2004)*, Santa Clara, California, USA, October 2004.

[33] J. Jeong, S. Kim, and A. Broad. Network reprogramming. TinyOS documentation, 2003. Visited 2006-04-06.
URL: http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf

[34] M. T. Jones. *TCP/IP Application Layer Protocols for Embedded Systems*. Charles River Media, June 2002.

[35] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs an d early experiences with ZebraNet. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 96–107, San Jose, California, 2002.

[36] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, Los Angeles, USA, April 2005.

[37] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Wireless Sensor Networks*, 2005.

[38] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *Proc. SenSys'05*, San Diego, CA, USA, November 2005.

[39] V. A. Kottapalli, A. S. Kiremidjian, J. P. Lynch, E. Carryer an d T. W. Kenny, K. H. Law, and Y. Lei. Two-tiered wireless sensor network architecture for structural health monitoring. In *Proceedings of the SPIE 10th Annual International Symposium on Smart Structures and Materials*, San Diego, CA, March 2000.

[40] Framework Labs. Protothreads for Objective-C/Cocoa. Visited 2006-04-06.
URL: http://www.frameworklabs.de/protothreads.html

[41] L. Laffea, R. Monson, R. Han, R. Manning, A. Glasser, S. Oncley, J. Sun, S. Burns, S. Semmer, and J. Militzer. Comprehensive monitoring of CO2 sequestration in subalpine forest ecosystems and its relation to global warming. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06)*, pages 423–424, Boulder, Colorado, USA, 2006.

[42] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[43] P. Levis, D. Gay, and D Culler. Active sensor networks. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005.

[44] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'04)*, San Francisco, California, USA, March 2004.

[45] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'04)*, March 2004.

[46] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proc. Second Intl. Conference on Mobile Systems, Applications and Services (MOBISYS 2004)*, June 2004.

[47] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[48] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.

[49] P. José Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *European Workshop on Wireless Sensor Networks*, Zurich, Switzerland, January 2006.

[50] A. J. Massa. *Embedded Software Development with eCos*. Prentice Hall, November 2002.

[51] W. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06)*, pages 167–180, Boulder, Colorado, USA, 2006.

[52] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[53] M. Melkonian. Get by Without an RTOS. *Embedded Systems Programming*, 13(10), September 2000.

[54] L. Mottola and G. Picco. Programming wireless sensor networks with logical neighborhoods. In *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks (InterSense '06)*, page 8, Nice, France, May 2006.

[55] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and scalable data dissemination service for wireless embedded devices. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, Miami, Florida, USA, December 2005.

[56] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *Proc. IPSN'05*, Los Angeles, CA, USA, April 2005.

[57] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks (DMSN '04)*, pages 78–87, Toronto, Canada, 2004.

[58] J. Paisley and J. Sventek. Real-time detection of grid bulk transfer traffic. In *Proceedings of the 10th IEEE/IFIP Network Operations Management Symposium*, Vancouver, Canada, April 2006.

[59] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

[60] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, San Diego, CA, USA, September 2003.

[61] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[62] J. Schiller, H. Ritter, A. Liers, and T. Voigt. Scatterweb - low power nodes and energy aware routing. In *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, USA, January 2005.

[63] V. Singhvi, A. Krause, C. Guestrin, Jr. J. Garrett, and S. Matthews. Intelligent light control using sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems (SenSys '05)*, pages 218–229, San Diego, California, USA, November 2005.

[64] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[65] J. T. Taylor. eXtreme Minimal Kernel. Shift-Right Technologies LLC. URL: http://www.shift-right.com/xmk/

[66] J. Turley. The Two Percent Solution. *Embedded Systems Design*, December 2002.

[67] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue (Kauai), Hawaii, USA, May 2003.

[68] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'04)*, San Francisco, California, USA, March 2004.

[69] M. Welsh, D. Myung, M. Gaynor, and S. Moulton. Resuscitation monitoring with a wireless sensor network. In *Circulation 108:1037: Journal of the American Heart Association, Resuscitation Science Symposium.*, October 2003.

[70] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation 2006*, Seattle, November 2006.

[71] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services (MobiSys '04)*, Boston, MA, USA, June 2004.

[72] Q. Xie, J. Liu, and P. H. Chou. Tapper: a lightweight scripting engine for highly constrained wireless sensor nodes. In *Proceedings of the fifth international conference on Information processing in sensor networks (IPSN '06), Poster session*, Nashville, Tennessee, USA, 2006.

II

Papers

Chapter 7

Paper A: Full TCP/IP for 8-Bit Architectures

Adam Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (ACM MobiSys 2003)*, San Francisco, USA, May 2003.

©2003 Usenix Association.

Abstract

We describe two small and portable TCP/IP implementations fulfilling the subset of RFC1122 requirements needed for full host-to-host interoperability. Our TCP/IP implementations do not sacrifice any of TCP's mechanisms such as urgent data or congestion control. They support IP fragment reassembly and the number of multiple simultaneous connections is limited only by the available RAM. Despite being small and simple, our implementations do not require their peers to have complex, full-size stacks, but can communicate with peers running a similarly light-weight stack. The code size is on the order of 10 kilobytes and RAM usage can be configured to be as low as a few hundred bytes.

7.1 Introduction

With the success of the Internet, the TCP/IP protocol suite has become a global standard for communication. TCP/IP is the underlying protocol used for web page transfers, e-mail transmissions, file transfers, and peer-to-peer networking over the Internet. For embedded systems, being able to run native TCP/IP makes it possible to connect the system directly to an intranet or even the global Internet. Embedded devices with full TCP/IP support will be first-class network citizens, thus being able to fully communicate with other hosts in the network.

Traditional TCP/IP implementations have required far too much resources both in terms of code size and memory usage to be useful in small 8 or 16-bit systems. Code size of a few hundred kilobytes and RAM requirements of several hundreds of kilobytes have made it impossible to fit the full TCP/IP stack into systems with a few tens of kilobytes of RAM and room for less than 100 kilobytes of code.

TCP [21] is both the most complex and the most widely used of the transport protocols in the TCP/IP stack. TCP provides reliable full-duplex byte stream transmission on top of the best-effort IP [20] layer. Because IP may reorder or drop packets between the sender and the receiver, TCP has to implement sequence numbering and retransmissions in order to achieve reliable, ordered data transfer.

We have implemented two small generic and portable TCP/IP implementations, *lwIP* (lightweight IP) and *uIP* (micro IP), with slightly different design goals. The lwIP implementation is a full-scale but simplified TCP/IP implementation that includes implementations of IP, ICMP, UDP and TCP and is modular enough to be easily extended with additional protocols. lwIP has support for multiple local network interfaces and has flexible configuration options which makes it suitable for a wide variety of devices.

The uIP implementation is designed to have only the absolute minimal set of features needed for a full TCP/IP stack. It can only handle a single network interface and does not implement UDP, but focuses on the IP, ICMP and TCP protocols.

Both implementations are fully written in the C programming language. We have made the source code available for both lwIP [7] and uIP [8]. Our implementations have been ported to numerous 8- and 16-bit platforms such as the AVR, H8S/300, 8051, Z80, ARM, M16c, and the x86 CPUs. Devices running our implementations have been used in numerous places throughout the Internet.

We have studied how the code size and RAM usage of a TCP/IP implementation affect the features of the TCP/IP implementation and the performance of the communication. We have limited our work to studying the implementation of TCP and IP protocols and the interaction between the TCP/IP stack and the application programs. Aspects such as address configuration, security, and energy consumption are out of the scope of this work.

The main contribution of our work is that we have shown that is it possible to implement a full TCP/IP stack that is small enough in terms of code size and memory usage to be useful even in limited 8-bit systems.

Recently, other small implementations of the TCP/IP stack have made it possible to run TCP/IP in small 8-bit systems. Those implementations are often heavily specialized for a particular application, usually an embedded web server, and are not suited for handling generic TCP/IP protocols. Future embedded networking applications such as peer-to-peer networking require that the embedded devices are able to act as first-class network citizens and run a TCP/IP implementation that is not tailored for any specific application.

Furthermore, existing TCP/IP implementations for small systems assume that the embedded device always will communicate with a full-scale TCP/IP implementation running on a workstation-class machine. Under this assumption, it is possible to remove certain TCP/IP mechanisms that are very rarely used in such situations. Many of those mechanisms are essential, however, if the embedded device is to communicate with another equally limited device, e.g., when running distributed peer-to-peer services and protocols.

This paper is organized as follows. After a short introduction to TCP/IP in Section 7.2, related work is presented in Section 7.3. Section 7.4 discusses RFC standards compliance. How memory and buffer management is done in our implementations is presented in Section 7.5 and the application program interface is discussed in Section 7.6. Details of the protocol implementations is given in Section 7.7 and Section 7.8 comments on the performance and maximum throughput of our implementations, presents throughput measurements from experiments and reports on the code size of our implementations. Section 7.9 gives ideas for future work. Finally, the paper is summarized and concluded in Section 7.10.

7.2 TCP/IP overview

From a high level viewpoint, the TCP/IP stack can be seen as a black box that takes incoming packets, and demultiplexes them between the currently active

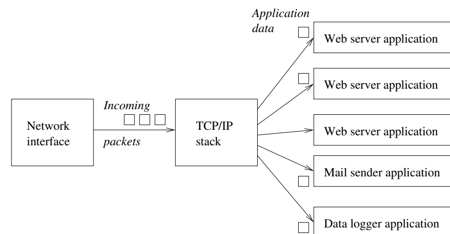

Figure 7.1: TCP/IP input processing.

connections. Before the data is delivered to the application, TCP sorts the packets so that they appear in the order they were sent. The TCP/IP stack will also send acknowledgments for the received packets.

Figure 7.1 shows how packets come from the network device, pass through the TCP/IP stack, and are delivered to the actual applications. In this example there are five active connections, three that are handled by a web server application, one that is handled by the e-mail sender application, and one that is handled by a data logger application.

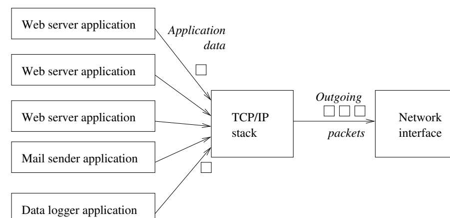

Figure 7.2: TCP/IP output processing.

A high level view of the output processing can be seen in Figure 7.2. The TCP/IP stack collects the data sent by the applications before it is actually sent onto the network. TCP has mechanisms for limiting the amount of data that is sent over the network, and each connection has a queue on which the data is held while waiting to be transmitted. The data is not removed from the queue until the receiver has acknowledged the reception of the data. If no acknowledgment is received within a specific time, the data is retransmitted.

Data arrives asynchronously from both the network and the application,

and the TCP/IP stack maintains queues in which packets are kept waiting for service. Because packets might be dropped or reordered by the network, incoming packets may arrive out of order. Such packets have to be queued by the TCP/IP stack until a packet that fills the gap arrives. Furthermore, because TCP limits the rate at which data that can be transmitted over each TCP connection, application data might not be immediately sent out onto the network.

The full TCP/IP suite consists of numerous protocols, ranging from low level protocols such as ARP which translates IP addresses to MAC addresses, to application level protocols such as SMTP that is used to transfer e-mail. We have concentrated our work on the TCP and IP protocols and will refer to upper layer protocols as “the application”. Lower layer protocols are often implemented in hardware or firmware and will be referred to as “the network device” that are controlled by the network device driver.

TCP provides a reliable byte stream to the upper layer protocols. It breaks the byte stream into appropriately sized segments and each segment is sent in its own IP packet. The IP packets are sent out on the network by the network device driver. If the destination is not on the physically connected network, the IP packet is forwarded onto another network by a router that is situated between the two networks. If the maximum packet size of the other network is smaller than the size of the IP packet, the packet is fragmented into smaller packets by the router. If possible, the size of the TCP segments are chosen so that fragmentation is minimized. The final recipient of the packet will have to reassemble any fragmented IP packets before they can be passed to higher layers.

7.3 Related work

There are numerous small TCP/IP implementations for embedded systems. The target architectures range from small 8-bit microcontrollers to 32-bit RISC architectures. Code size varies from a few kilobytes to hundreds of kilobytes. RAM requirements can be as low as 10 bytes up to several megabytes.

Existing TCP/IP implementations can roughly be divided into two categories; those that are adaptations of the Berkeley BSD TCP/IP implementation [18], and those that are written independently from the BSD code. The BSD implementation was originally written for workstation-class machines and was not designed for the limitations of small embedded systems. Because of that, implementations that are derived from the BSD code base are usually suited for larger architectures than our target. An example of a BSD-

derived implementation is the InterNiche NicheStack [11], which needs around 50 kilobytes of code space on a 32-bit ARM system.

Many of the independent TCP/IP implementations for embedded processors use a simplified model of the TCP/IP stack which makes several assumptions about the communication environment. The most common assumption is that the embedded system always will communicate with a system such as a PC that runs a full scale, standards compliant TCP/IP implementation. By relying on the standards compliance of the remote host, even an extremely simplified, uncompliant, TCP/IP implementation will be able to communicate. The communication may very well fail, however, once the system is to communicate with another simplified TCP/IP implementation such as another embedded system of the same kind. We will briefly cover a number of such simplifications that are used by existing implementations.

One usual simplification is to tailor the TCP/IP stack for a specific application such as a web server. By doing this, only the parts of the TCP/IP protocols that are required by the application need to be implemented. For instance, a web server application does not need support for urgent data and does not need to actively open TCP connections to other hosts. By removing those mechanisms from the implementation, the complexity is reduced.

The smallest TCP/IP implementations in terms of RAM and code space requirements are heavily specialized for serving web pages and use an approach where the web server does not hold any connection state at all. For example, the iPic match-head sized server [26] and Jeremy Bentham's PICmicro stack [1] require only a few tens of bytes of RAM to serve simple web pages. In such an implementation, retransmissions cannot be made by the TCP module in the embedded system because nothing is known about the active connections. In order to achieve reliable transfers, the system has to rely on the remote host to perform retransmissions. It is possible to run a very simple web server with such an implementation, but there are serious limitations such as not being able to serve web pages that are larger than the size of a single TCP segment, which typically is about one kilobyte.

Other TCP/IP implementations such as the Atmel TCP/IP stack [5] save code space by leaving out certain vital TCP mechanisms. In particular, they often leave out TCP's congestion control mechanisms, which are used to reduce the sending rate when the network is overloaded. While an implementation with no congestion control might work well when connected to a single Ethernet segment, problems can arise when communication spans several networks. In such cases, the intermediate nodes such as switches and routers may be overloaded. Because congestion primarily is caused by the amount of pack-

ets in the network, and not the size of these packets, even small 8-bit systems are able to produce enough traffic to cause congestion. A TCP/IP implementation lacking congestion control mechanisms should not be used over the global Internet as it might contribute to congestion collapse [9].

Texas Instrument's MSP430 TCP/IP stack [6] and the TinyTCP code [4] use another common simplification in that they can handle only one TCP connection at a time. While this is a sensible simplification for many applications, it seriously limits the usefulness of the TCP/IP implementation. For example, it is not possible to communicate with two simultaneous peers with such an implementation. The CMX Micronet stack [27] uses a similar simplification in that it sets a hard limit of 16 on the maximum number of connections.

Yet another simplification that is used by LiveDevices Embedinet implementation [12] and others is to disregard the maximum segment size that a receiver is prepared to handle. Instead, the implementation will send segments that fit into an Ethernet frame of 1500 bytes. This works in a lot of cases due to the fact that many hosts are able to receive packets that are 1500 bytes or larger. Communication will fail, however, if the receiver is a system with limited memory resources that is not able to handle packets of that size.

Finally, the most common simplification is to leave out support for reassembling fragmented IP packets. Even though fragmented IP packets are quite infrequent [25], there are situations in which they may occur. If packets travel over a path which fragments the packets, communication is impossible if the TCP/IP implementation is unable to correctly reassemble them. TCP/IP implementations that are able to correctly reassemble fragmented IP packets, such as the Kadak KwikNET stack [22], are usually too large in terms of code size and RAM requirements to be practical for 8-bit systems.

7.4 RFC-compliance

The formal requirements for the protocols in the TCP/IP stack is specified in a number of RFC documents published by the Internet Engineering Task Force, IETF. Each of the protocols in the stack is defined in one more RFC documents and RFC1122 [2] collects all requirements and updates the previous RFCs.

The RFC1122 requirements can be divided into two categories; those that deal with the host to host communication and those that deal with communication between the application and the networking stack. An example of the first kind is *"A TCP MUST be able to receive a TCP option in any segment"* and an example of the second kind is *"There MUST be a mechanism for reporting soft*

Table 7.1: TCP/IP features implemented by uIP and lwIP

Feature	uIP	lwIP
IP and TCP checksums	x	x
IP fragment reassembly	x	x
IP options		
Multiple interfaces		x
UDP		x
Multiple TCP connections	x	x
TCP options	x	x
Variable TCP MSS	x	x
RTT estimation	x	x
TCP flow control	x	x
Sliding TCP window		x
TCP congestion control	Not needed	x
Out-of-sequence TCP data		x
TCP urgent data	x	x
Data buffered for rexmit		x

TCP error conditions to the application." A TCP/IP implementation that violates requirements of the first kind may not be able to communicate with other TCP/IP implementations and may even lead to network failures. Violation of the second kind of requirements will only affect the communication within the system and will not affect host-to-host communication.

In our implementations, we have implemented all RFC requirements that affect host-to-host communication. However, in order to reduce code size, we have removed certain mechanisms in the interface between the application and the stack, such as the soft error reporting mechanism and dynamically configurable type-of-service bits for TCP connections. Since there are only very few applications that make use of those features, we believe that they can be removed without loss of generality. Table 7.1 lists the features that uIP and lwIP implements.

7.5 Memory and buffer management

In our target architecture, RAM is the most scarce resource. With only a few kilobytes of RAM available for the TCP/IP stack to use, mechanisms used in

traditional TCP/IP cannot be directly applied.

Because of the different design goals for the lwIP and the uIP implementations, we have chosen two different memory management solutions. The lwIP implementation has dynamic buffer and memory allocation mechanisms where memory for holding connection state and packets is dynamically allocated from a global pool of available memory blocks. Packets are contained in one or more dynamically allocated buffers of fixed size. The size of the packet buffers is determined by a configuration option at compile time. Buffers are allocated by the network device driver when an incoming packet arrives. If the packet is larger than one buffer, more buffers are allocated and the packet is split into the buffers. If the incoming packet is queued by higher layers of the stack or the application, a reference counter in the buffer is incremented. The buffer will not be deallocated until the reference count is zero.

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding packets and has a fixed table for holding connection state. The global packet buffer is large enough to contain one packet of maximum size. When a packet arrives from the network, the device driver places it in the global buffer and calls the TCP/IP stack. If the packet contains data, the TCP/IP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into a secondary buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data. Packets that arrive when the application is processing the data must be queued, either by the network device or by the device driver. Most single-chip Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames. Devices that are handled by the processor, such as RS-232 ports, can copy incoming bytes to a separate buffer during application processing. If the buffers are full, the incoming packet is dropped. This will cause performance degradation, but only when multiple connections are running in parallel. This is because uIP advertises a very small receiver window, which means that only a single TCP segment will be in the network per connection.

Outgoing data is also handled differently because of the different buffer schemes. In lwIP, an application that wishes to send data passes the length and a pointer to the data to the TCP/IP stack as well as a flag which indicates whether the data is volatile or not. The TCP/IP stack allocates buffers of suitable size and, depending on the volatile flag, either copies the data into the buffers or references the data through pointers. The allocated buffers contain space for the TCP/IP stack to prepend the TCP/IP and link layer headers.

After the headers are written, the stack passes the buffers to the network device driver. The buffers are not deallocated when the device driver is finished sending the data, but held on a retransmission queue. If the data is lost in the network and have to be retransmitted, the buffers on retransmission queue will be retransmitted. The buffers are not deallocated until the data is known to be received by the peer. If the connection is aborted because of an explicit request from the local application or a reset segment from the peer, the connection's buffers are deallocated.

In uIP, the same global packet buffer that is used for incoming packets is also used for the TCP/IP headers of outgoing data. If the application sends dynamic data, it may use the parts of the global packet buffer that are not used for headers as a temporary storage buffer. To send the data, the application passes a pointer to the data as well as the length of the data to the stack. The TCP/IP headers are written into the global buffer and once the headers have been produced, the device driver sends the headers and the application data out on the network. The data is not queued for retransmissions. Instead, the application will have to reproduce the data if a retransmission is necessary.

The total amount of memory usage for our implementations depends heavily on the applications of the particular device in which the implementations are to be run. The memory configuration determines both the amount of traffic the system should be able to handle and the maximum amount of simultaneous connections. A device that will be sending large e-mails while at the same time running a web server with highly dynamic web pages and multiple simultaneous clients, will require more RAM than a simple Telnet server. It is possible to run the uIP implementation with as little as 200 bytes of RAM, but such a configuration will provide extremely low throughput and will only allow a small number of simultaneous connections.

7.6 Application program interface

The Application Program Interface (API) defines the way the application program interacts with the TCP/IP stack. The most commonly used API for TCP/IP is the BSD socket API which is used in most Unix systems and has heavily influenced the Microsoft Windows WinSock API. Because the socket API uses stop-and-wait semantics, it requires support from an underlying multitasking operating system. Since the overhead of task management, context switching and allocation of stack space for the tasks might be too high in our target architecture, the BSD socket interface is not suitable for our purposes.

Instead, we have chosen an event driven interface where the application is invoked in response to certain events. Examples of such events are data arriving on a connection, an incoming connection request, or a poll request from the stack. The event based interface fits well in the event based structure used by operating systems such as TinyOS [10]. Furthermore, because the application is able to act on incoming data and connection requests as soon as the TCP/IP stack receives the packet, low response times can be achieved even in low-end systems.

7.7 Protocol implementations

The protocols in the TCP/IP protocol suite are designed in a layered fashion where each protocol performs a specific function and the interactions between the protocol layers are strictly defined. While the layered approach is a good way to design protocols, it is not always the best way to implement them. For the lwIP implementation, we have chosen a fully modular approach where each protocol implementation is kept fairly separate from the others. In the smaller uIP implementation, the protocol implementations are tightly coupled in order to save code space.

7.7.1 Main control loop

The lwIP and uIP stacks can be run either as a task in a multitasking system, or as the main program in a singletasking system. In both cases, the main control loop (Figure 7.3) does two things repeatedly:

1. Check if a packet has arrived from the network.

2. Check if a periodic timeout has occurred.

If a packet has arrived, the input handler of the TCP/IP stack is invoked. The input handler function will never block, but will return at once. When it returns, the stack or the application for which the incoming packet was intended may have produced one or more reply packets which should be sent out. If so, the network device driver is called to send out these packets.

Periodic timeouts are used to drive TCP mechanisms that depend on timers, such as delayed acknowledgments, retransmissions and round-trip time estimations. When the main control loop infers that the periodic timer should fire, it invokes the timer handler of the TCP/IP stack. Because the TCP/IP stack may

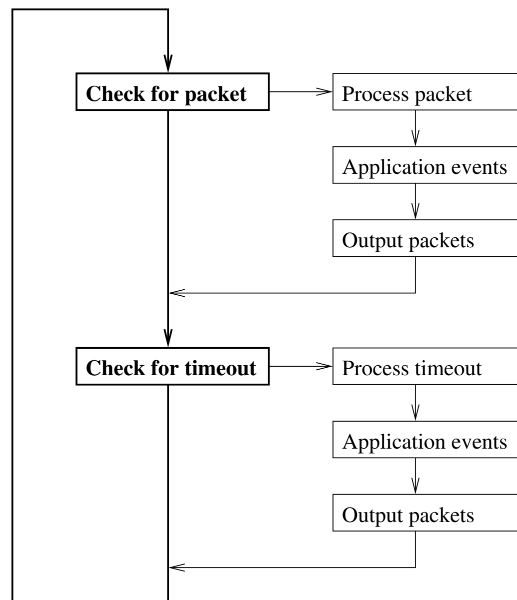

Figure 7.3: The main control loop.

perform retransmissions when dealing with a timer event, the network device driver is called to send out the packets that may have been produced.

This is similar to how the BSD implementations drive the TCP/IP stack, but BSD uses software interrupts and a task scheduler to initiate input handlers and timers. In our limited system, we do not depend on such mechanisms being available.

7.7.2 IP — Internet Protocol

When incoming packets are processed by lwIP and uIP, the IP layer is the first protocol that examines the packet. The IP layer does a few simple checks such as if the destination IP address of the incoming packet matches any of the local IP address and verifies the IP header checksum. Since there are no IP options that are strictly required and because they are very uncommon, both lwIP and uIP drop any IP options in received packets.

IP fragment reassembly

In both lwIP and uIP, IP fragment reassembly is implemented using a separate buffer that holds the packet to be reassembled. An incoming fragment is copied into the right place in the buffer and a bit map is used to keep track of which fragments have been received. Because the first byte of an IP fragment is aligned on an 8-byte boundary, the bit map requires a small amount of memory. When all fragments have been reassembled, the resulting IP packet is passed to the transport layer. If all fragments have not been received within a specified time frame, the packet is dropped.

The current implementation only has a single buffer for holding packets to be reassembled, and therefore does not support simultaneous reassembly of more than one packet. Since fragmented packets are uncommon, we belive this to be a reasonable decision. Extending our implementation to support multiple buffers would be straightforward, however.

Broadcasts and multicasts

IP has the ability to broadcast and multicast packets on the local network. Such packets are addressed to special broadcast and multicast addresses. Broadcast is used heavily in many UDP based protocols such as the Microsoft Windows file-sharing SMB protocol. Multicast is primarily used in protocols used for multimedia distribution such as RTP. TCP is a point-to-point protocol and does not use broadcast or multicast packets.

Because lwIP supports applications using UDP, it has support for both sending and receiving broadcast and multicast packets. In contrast, uIP does not have UDP support and therefore handling of such packets has not been implemented.

7.7.3 ICMP — Internet Control Message Protocol

The ICMP protocol is used for reporting soft error conditions and for querying host parameters. Its main use is, however, the echo mechanism which is used by the `ping` program.

The ICMP implementations in lwIP and uIP are very simple as we have restricted them to only implement ICMP echo messages. Replies to echo messages are constructed by simply swapping the source and destination IP addresses of incoming echo requests and rewriting the ICMP header with the Echo-Reply message type. The ICMP checksum is adjusted using standard techniques [23].

Since only the ICMP echo message is implemented, there is no support for Path MTU discovery or ICMP redirect messages. Neither of these is strictly required for interoperability; they are performance enhancement mechanisms.

7.7.4 TCP — Transmission Control Protocol

The TCP implementations in lwIP and uIP are driven by incoming packets and timer events. IP calls TCP when a TCP packet arrives and the main control loop calls TCP periodically.

Incoming packets are parsed by TCP and if the packet contains data that is to be delivered to the application, the application is invoked by the means of a function call. If the incoming packet acknowledges previously sent data, the connection state is updated and the application is informed, allowing it to send out new data.

Listening connections

TCP allows a connection to listen for incoming connection requests. In our implementations, a listening connection is identified by the 16-bit port number and incoming connection requests are checked against the list of listening connections. This list of listening connections is dynamic and can be altered by the applications in the system.

Sending data

When sending data, an application will have to check the number of available bytes in the send window and adjust the number of bytes to send accordingly. The size of the send window is dictated by the memory configuration as well as the buffer space announced by the receiver of the data. If no buffer space is available, the application has to defer the send and wait until later.

Buffer space becomes available when an acknowledgment from the receiver of the data has been received. The stack informs the application of this event, and the application may then repeat the sending procedure.

Sliding window

Most TCP implementations use a sliding window mechanism for sending data. Multiple data segments are sent in succession without waiting for an acknowledgment for each segment.

The sliding window algorithm uses a lot of 32-bit operations and because 32-bit arithmetic is fairly expensive on most 8-bit CPUs, uIP does not implement it. Also, uIP does not buffer sent packets and a sliding window implementation that does not buffer sent packets will have to be supported by a complex application layer. Instead, uIP allows only a single TCP segment per connection to be unacknowledged at any given time. lwIP, on the other hand, implements TCP's sliding window mechanism using output buffer queues and therefore does not add additional complexity to the application layer.

It is important to note that even though most TCP implementations use the sliding window algorithm, it is not required by the TCP specifications. Removing the sliding window mechanism does not affect interoperability in any way.

Round-trip time estimation

TCP continuously estimates the current Round-Trip Time (RTT) of every active connection in order to find a suitable value for the retransmission time-out.

We have implemented the RTT estimation using TCP's periodic timer. Each time the periodic timer fires, it increments a counter for each connection that has unacknowledged data in the network. When an acknowledgment is received, the current value of the counter is used as a sample of the RTT. The sample is used together with the standard TCP RTT estimation function [13] to calculate an estimate of the RTT. Karn's algorithm [14] is used to ensure that retransmissions do not skew the estimates.

Retransmissions

Retransmissions are driven by the periodic TCP timer. Every time the periodic timer is invoked, the retransmission timer for each connection is decremented. If the timer reaches zero, a retransmission should be made.

The actual retransmission operation is handled differently in uIP and in lwIP. lwIP maintains two output queues: one holds segments that have not yet been sent, the other holds segments that have been sent but not yet been acknowledged by the peer. When a retransmission is required, the first segment on the queue of segments that has not been acknowledged is sent. All other segments in the queue are moved to the queue with unsent segments.

As uIP does not keep track of packet contents after they have been sent by the device driver, uIP requires that the application takes an active part in performing the retransmission. When uIP decides that a segment should be re-

transmitted, it calls the application with a flag set indicating that a retransmission is required. The application checks the retransmission flag and produces the same data that was previously sent. From the application's standpoint, performing a retransmission is not different from how the data originally was sent. Therefore the application can be written in such a way that the same code is used both for sending data and retransmitting data. Also, it is important to note that even though the actual retransmission operation is carried out by the application, it is the responsibility of the stack to know when the retransmission should be made. Thus the complexity of the application does not necessarily increase because it takes an active part in doing retransmissions.

Flow control

The purpose of TCP's flow control mechanisms is to allow communication between hosts with wildly varying memory dimensions. In each TCP segment, the sender of the segment indicates its available buffer space. A TCP sender must not send more data than the buffer space indicated by the receiver.

In our implementations, the application cannot send more data than the receiving host can buffer. Before sending data, the application checks how many bytes it is allowed to send and does not send more data than the other host can accept. If the remote host cannot accept any data at all, the stack initiates the zero window probing mechanism.

The application is responsible for controlling the size of the window size indicated in sent segments. If the application must wait or buffer data, it can explicitly close the window so that the sender will not send data until the application is able to handle it.

Congestion control

The congestion control mechanisms limit the number of simultaneous TCP segments in the network. The algorithms used for congestion control [13] are designed to be simple to implement and require only a few lines of code.

Since uIP only handles one in-flight TCP segment per connection, the amount of simultaneous segments cannot be further limited, thus the congestion control mechanisms are not needed. lwIP has the ability to have multiple in-flight segments and therefore implements all of TCP's congestion control mechanisms.

Urgent data

TCP's urgent data mechanism provides an application-to-application notification mechanism, which can be used by an application to mark parts of the data stream as being more urgent than the normal stream. It is up to the receiving application to interpret the meaning of the urgent data.

In many TCP implementations, including the BSD implementation, the urgent data feature increases the complexity of the implementation because it requires an asynchronous notification mechanism in an otherwise synchronous API. As our implementations already use an asynchronous event based API, the implementation of the urgent data feature does not lead to increased complexity.

Connection state

Each TCP connection requires a certain amount of state information in the embedded device. Because the state information uses RAM, we have aimed towards minimizing the amount of state needed for each connection in our implementations.

The uIP implementation, which does not use the sliding window mechanism, requires far less state information than the lwIP implementation. The sliding window implementation requires that the connection state includes several 32-bit sequence numbers, not only for keeping track of the current sequence numbers of the connection, but also for remembering the sequence numbers of the last window updates. Furthermore, because lwIP is able to handle multiple local IP addresses, the connection state must include the local IP address. Finally, as lwIP maintains queues for outgoing segments, the memory for the queues is included in the connection state. This makes the state information needed for lwIP nearly 60 bytes larger than that of uIP which requires 30 bytes per connection.

7.8 Results

7.8.1 Performance limits

In TCP/IP implementations for high-end systems, processing time is dominated by the checksum calculation loop, the operation of copying packet data and context switching [15]. Operating systems for high-end systems often have multiple protection domains for protecting kernel data from user processes and

user processes from each other. Because the TCP/IP stack is run in the kernel, data has to be copied between the kernel space and the address space of the user processes and a context switch has to be performed once the data has been copied. Performance can be enhanced by combining the copy operation with the checksum calculation [19]. Because high-end systems usually have numerous active connections, packet demultiplexing is also an expensive operation [17].

A small embedded device does not have the necessary processing power to have multiple protection domains and the power to run a multitasking operating system. Therefore there is no need to copy data between the TCP/IP stack and the application program. With an event based API there is no context switch between the TCP/IP stack and the applications.

In such limited systems, the TCP/IP processing overhead is dominated by the copying of packet data from the network device to host memory, and checksum calculation. Apart from the checksum calculation and copying, the TCP processing done for an incoming packet involves only updating a few counters and flags before handing the data over to the application. Thus an estimate of the CPU overhead of our TCP/IP implementations can be obtained by calculating the amount of CPU cycles needed for the checksum calculation and copying of a maximum sized packet.

7.8.2 The impact of delayed acknowledgments

Most TCP receivers implement the delayed acknowledgment algorithm [3] for reducing the number of pure acknowledgment packets sent. A TCP receiver using this algorithm will only send acknowledgments for every other received segment. If no segment is received within a specific time-frame, an acknowledgment is sent. The time-frame can be as high as 500 ms but typically is 200 ms.

A TCP sender such as uIP that only handles a single outstanding TCP segment will interact poorly with the delayed acknowledgment algorithm. Because the receiver only receives a single segment at a time, it will wait as much as 500 ms before an acknowledgment is sent. This means that the maximum possible throughput is severely limited by the 500 ms idle time.

Thus the maximum throughput equation when sending data from uIP will be $p = s/(t+t_d)$ where s is the segment size and t_d is the delayed acknowledgment timeout, which typically is between 200 and 500 ms. With a segment size of 1000 bytes, a round-trip time of 40 ms and a delayed acknowledgment timeout of 200 ms, the maximum throughput will be 4166 bytes per second. With

the delayed acknowledgment algorithm disabled at the receiver, the maximum throughput would be 25000 bytes per second.

It should be noted, however, that since small systems running uIP are not very likely to have large amounts of data to send, the delayed acknowledgment throughput degradation of uIP need not be very severe. Small amounts of data sent by such a system will not span more than a single TCP segment, and would therefore not be affected by the throughput degradation anyway.

The maximum throughput when uIP acts as a receiver is not affected by the delayed acknowledgment throughput degradation.

7.8.3 Measurements

For our experiments we connected a 450 MHz Pentium III PC running FreeBSD 4.7 to an Ethernut board [16] through a dedicated 10 megabit/second Ethernet network. The Ethernut board is a commercially available embedded system equipped with a RealTek RTL8019AS Ethernet controller, an Atmel Atmega128 AVR microcontroller running at 14.7456 MHz with 128 kilobytes of flash ROM for code storage and 32 kilobytes of RAM. The FreeBSD host was configured to run the Dummynet delay emulator software [24] in order to facilitate controlled delays for the communication between the PC and the embedded system.

In the embedded system, a simple web server was run on top of the uIP and lwIP stacks. Using the `fetch` file retrieval utility, a file consisting of null bytes was downloaded ten times from the embedded system. The reported throughput was logged, and the mean throughput of the ten downloads was calculated. By redirecting file output to `/dev/null`, the file was immediately discarded by the FreeBSD host. The file size was 200 kilobytes for the uIP tests, and 200 megabytes for the lwIP tests. The size of the file made it impossible to keep it all in the memory of the embedded system. Instead, the file was generated by the web server as it was sent out on the network.

The total TCP/IP memory consumption in the embedded system was varied by changing the send window size. For uIP, the send window was varied between 50 bytes and the maximum possible value of 1450 bytes in steps of 50 bytes. The send window configuration translates into a total RAM usage of between 400 bytes and 3 kilobytes. The lwIP send window was varied between 500 and 11000 bytes in steps of 500 bytes, leading to a total RAM consumption of between 5 and 16 kilobytes.

Figure 7.4 shows the mean throughput of the ten file downloads from the web server running on top of uIP, with an additional 10 ms delay created by

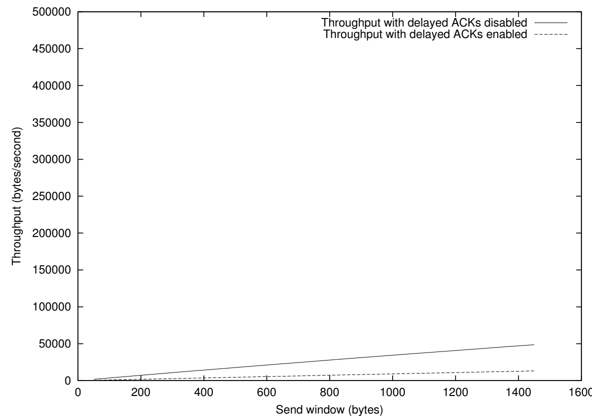

Figure 7.4: uIP sending data with 10 ms emulated delay.

the Dummynet delay emulator. The two curves show the measured throughput with the delayed acknowledgment algorithm disabled and enabled at the receiving FreeBSD host, respectively. The performance degradation caused by the delayed acknowledgments is evident.

Figure 7.5 shows the same setup, but without the 10 ms emulated delay. The lower curve, showing the throughput with delayed acknowledgments enabled, is very similar to the lower one in Figure 7.4. The upper curve, however, does not show the same linear relation as the previous figure, but shows an increasing throughput where the increase declines with increasing send window size. One explanation for the declining increase of throughput is that the round-trip time increases with the send window size because of the increased per-packet processing time. Figure 7.6 shows the round-trip time as a function of packet size. These measurements were taken using the `ping` program and therefore include the cost for the packet copying operation twice; once for packet input and once for packet output.

The throughput of lwIP shows slightly different characteristics. Figure 7.7 shows three measured throughput curves, without emulated delay, and with emulated delays of 10 ms and 20 ms. For all measurements, the delayed acknowledgment algorithm is enabled at the FreeBSD receiver. We see that for small send window sizes, lwIP also suffers from the delayed acknowledgment throughput degradation. With a send window larger than two maximum TCP

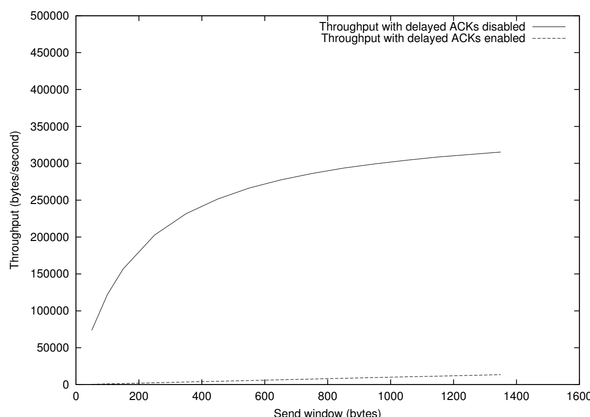

Figure 7.5: uIP sending data without emulated delay.

segment sizes (3000 bytes), lwIP is able to send out two TCP segments per round-trip time and thereby avoids the delayed acknowledgments throughput degradation. Without emulated delay, the throughput quickly reaches a maximum of about 415 kilobytes per second. This limit is likely to be the processing limit of the lwIP code in the embedded system and therefore is the maximum possible throughput for lwIP in this particular system.

The maximum throughput with emulated delays is lower than without delay emulation, and the similarity of the two curves suggests that the throughput degradation could be caused by interaction with the Dummynet software.

7.8.4 Code size

The code was compiled for the 32-bit Intel x86 and the 8-bit Atmel AVR platforms using gcc [28] versions 2.95.3 and 3.3 respectively, with code size optimization turned on. The resulting size of the compiled code can be seen in Tables 7.2 to 7.5. Even though both implementations support ARP and SLIP and lwIP includes UDP, only the protocols discussed in this paper are presented. Because the protocol implementations in uIP are tightly coupled, the individual sizes of the implementations are not reported.

There are several reasons for the dramatic difference in code size between lwIP and uIP. In order to support the more complex and configurable TCP im-

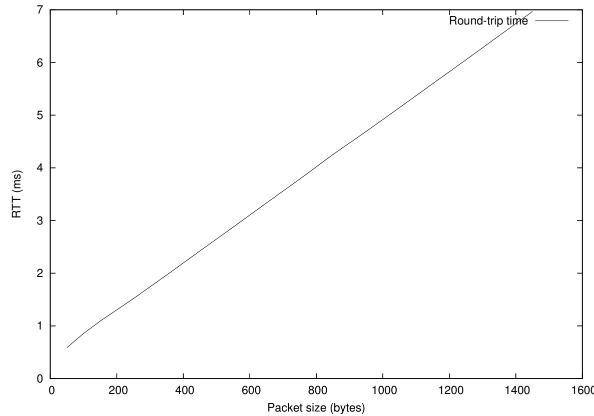

Figure 7.6: Round-trip time as a function of packet size.

Table 7.2: Code size for uIP (x86)

Function	Code size (bytes)
Checksumming	464
IP, ICMP and TCP	4724
Total	5188

plementation, lwIP has significantly more complex buffer and memory management than uIP. Since lwIP can handle packets that span several buffers, the checksum calculation functions in lwIP are more complex than those in uIP. The support for dynamically changing network interfaces in lwIP also contributes to the size increase of the IP layer because the IP layer has to manage multiple local IP addresses. The IP layer in lwIP is further made larger by the fact that lwIP has support for UDP, which requires that the IP layer is able handle broadcast and multicast packets. Likewise, the ICMP implementation in lwIP has support for UDP error messages which have not been implemented in uIP.

The TCP implementation is lwIP is nearly twice as large as the full IP, ICMP and TCP implementation in uIP. The main reason for this is that lwIP implements the sliding window mechanism which requires a large amount of

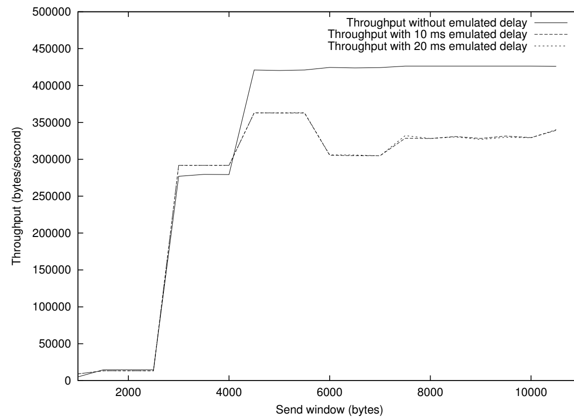

Figure 7.7: lwIP sending data with and without emulated delays.

Table 7.3: Code size for uIP (AVR)

Function	Code size (bytes)
Checksumming	712
IP, ICMP and TCP	4452
Total	5164

buffer and queue management functionality that is not required in uIP.

The different memory and buffer management schemes used by lwIP and uIP have implications on code size, mainly in 8-bit systems. Because uIP uses a global buffer for all incoming packets, the absolute memory addresses of the protocol header fields are known at compile time. Using this information, the compiler is able to generate code that uses absolute addressing, which on many 8-bit processors requires less code than indirect addressing.

Is it interesting to note that the size of the compiled lwIP code is larger on the AVR than on the x86, while the uIP code is of about the same size on the two platforms. The main reason for this is that lwIP uses 32-bit arithmetic to a much larger degree than uIP and each 32-bit operation is compiled into a large number of machine code instructions.

Table 7.4: Code size for lwIP (x86)

Function	Code size (bytes)
Memory management	2512
Checksumming	504
Network interfaces	364
IP	1624
ICMP	392
TCP	9192
Total	14588

Table 7.5: Code size for lwIP (AVR)

Function	Code size (bytes)
Memory management	3142
Checksumming	1116
Network interfaces	458
IP	2216
ICMP	594
TCP	14230
Total	21756

7.9 Future work

Prioritized connections. It is advantageous to be able to prioritize certain connections such as Telnet connections for manual configuration of the device. Even in a system that is under heavy load from numerous clients, it should be possible to remotely control and configure the device. In order to do provide this, different connection types could be given different priority. For efficiency, such differentiation should be done as far down in the system as possible, preferably in the device driver.

Security aspects. When connecting systems to a network, or even to the global Internet, the security of the system is very important. Identifying levels of security and mechanisms for implementing security for embedded devices is crucial for connecting systems to the global Internet.

Address auto-configuration. If hundreds or even thousands of small em-

bedded devices should be deployed, auto-configuration of IP addresses is advantageous. Such mechanisms already exist in IPv6, the next version of the Internet Protocol, and are currently being standardized for IPv4.

Improving throughput. The throughput degradation problem caused by the poor interaction with the delayed acknowledgment algorithm should be fixed. By increasing the maximum number of in-flight segments from one to two, the problem will not appear. When increasing the amount of in-flight segments, congestion control mechanisms will have to be employed. Those mechanisms are trivial, however, when the upper limit is two simultaneous segments.

Performance enhancing proxy. It might be possible to increase the performance of communication with the embedded devices through the use of a proxy situated near the devices. Such a proxy would have more memory than the devices and could assume responsibility for buffering data.

7.10 Summary and conclusions

We have shown that it is possible to fit a full scale TCP/IP implementation well within the limits of an 8-bit microcontroller, but that the throughput of such a small implementation will suffer. We have not removed any TCP/IP mechanisms in our implementations, but have full support for reassembly of IP fragments and urgent TCP data. Instead, we have minimized the interface between the TCP/IP stack and the application.

The maximum achievable throughput for our implementations is determined by the send window size that the TCP/IP stack has been configured to use. When sending data with uIP, the delayed ACK mechanism at the receiver lowers the maximum achievable throughput considerably. In many situations however, a limited system running uIP will not produce so much data that this will cause problems. lwIP is not affected by the delayed ACK throughput degradation when using a large enough send window.

7.11 Acknowledgments

Many thanks go to Martin Nilsson, who has provided encouragement and been a source of inspiration throughout the preparation of this paper. Thanks also go to Deborah Wallach for comments and suggestions, the anonymous reviewers whose comments were highly appreciated, and to all who have contributed bugfixes, patches and suggestions to the lwIP and uIP implementations.

Bibliography

[1] J. Bentham. *TCP/IP Lean: Web servers for embedded systems*. CMP Books, October 2000.

[2] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, Internet Engineering Task Force, October 1989.

[3] D. D. Clark. Window and acknowledgement strategy in TCP. RFC 813, Internet Engineering Task Force, July 1982.

[4] G. H. Cooper. TinyTCP. Web page. 2002-10-14.
URL: http://www.csonline.net/bpaddock/tinytcp/

[5] Atmel Corporation. Embedded web server. AVR 460, January 2001. Avalible from www.atmel.com.

[6] A. Dannenberg. MSP430 internet connectivity. SLAA 137, November 2001. Avalible from www.ti.com.

[7] A. Dunkels. lwIP - a lightweight TCP/IP stack. Web page. 2002-10-14.
URL: http://www.sics.se/~adam/lwip/

[8] A. Dunkels. uIP - a TCP/IP stack for 8- and 16-bit microcontrollers. Web page. 2003-10-21.
URL: http://dunkels.com/adam/uip/

[9] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, August 1999.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming*

Languages and Operating Systems, Cambridge, Massachusets, USA, November 2000.

[11] InterNiche Technologies Inc. NicheStack portable TCP/IP stack. Web page. 2002-10-14.
URL: http://www.iniche.com/products/tcpip.htm

[12] LiveDevices Inc. Embedinet - embedded internet software products. Web page. 2002-10-14.
URL: http://www.livedevices.com/net_products/embedinet.shtml

[13] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Conference*, Stanford, California, August 1988.

[14] P. Karn and C. Partridge. Improving round-trip time estimates in reliablie transport protocols. In *Proceedings of the SIGCOMM '87 Conference*, Stowe, Vermont, August 1987.

[15] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM '93 Symposium*, pages 259–268, September 1993.

[16] H. Kipp. Ethernut embedded ethernet. Web page. 2002-10-14.
URL: http://www.ethernut.de/en/

[17] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–279, Baltimore, Maryland, August 1992.

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[19] C. Partridge and S. Pink. A faster UDP. *IEEE/ACM Transactions in Networking*, 1(4):429–439, August 1993.

[20] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[21] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[22] Kadak Products. Kadak KwikNET TCP/IP stack. Web page. 2002-10-14.
URL: http://www.kadak.com/html/kdkp1030.htm

[23] A. Rijsinghani. Computation of the internet checksum via incremental update. RFC 1624, Internet Engineering Task Force, May 1994.

[24] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[25] C. Shannon, D. Moore, and K. Claffy. Beyond folklore: Observations on fragmented traffic. *IEEE/ACM Transactions on Networking*, 10(6), December 2002.

[26] H. Shrikumar. IPic - a match head sized web-server. Web page. 2002-10-14.
URL: http://www-ccs.cs.umass.edu/~shri/iPic.html

[27] CMX Systems. CMX-MicroNet true TCP/IP networking. Web page. 2002-10-14.
URL: http://www.cmx.com/micronet.htm

[28] The GCC Team. The GNU compiler collection. Web page. 2002-10-14.
URL: http://gcc.gnu.org/

Chapter 8

Paper B: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors

Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE Emnets 2004)*, Tampa, Florida, USA,

©2004 Institute of Electrical and Electronics Engineers.

Abstract

Wireless sensor networks are composed of large numbers of tiny networked devices that communicate untethered. For large scale networks it is important to be able to dynamically download code into the network. In this paper we present Contiki, a lightweight operating system with support for dynamic loading and replacement of individual programs and services. Contiki is built around an event-driven kernel but provides optional preemptive multi-threading that can be applied to individual processes. We show that dynamic loading and unloading is feasible in a resource constrained environment, while keeping the base system lightweight and compact.

8.1 Introduction

Wireless sensor networks are composed of large numbers of tiny sensor devices with wireless communication capabilities. The sensor devices autonomously form networks through which sensor data is transported. The sensor devices are often severely resource constrained. An on-board battery or solar panel can only supply limited amounts of power. Moreover, the small physical size and low per-device cost limit the complexity of the system. Typical sensor devices [1, 2, 5] are equipped with 8-bit microcontrollers, code memory on the order of 100 kilobytes, and less than 20 kilobytes of RAM. Moore's law predicts that these devices can be made significantly smaller and less expensive in the future. While this means that sensor networks can be deployed to greater extents, it does not necessarily imply that the resources will be less constrained.

For the designer of an operating system for sensor nodes, the challenge lies in finding lightweight mechanisms and abstractions that provide a rich enough execution environment while staying within the limitations of the constrained devices. We have developed Contiki, an operating system developed for such constrained environments. Contiki provides dynamic loading and unloading of individual programs and services. The kernel is event-driven, but the system supports preemptive multi-threading that can be applied on a per-process basis. Preemptive multi-threading is implemented as a library that is linked only with programs that explicitly require multi-threading.

Contiki is implemented in the C language and has been ported to a number of microcontroller architectures, including the Texas Instruments MSP430 and the Atmel AVR. We are currently running it on the ESB platform [5]. The ESB uses the MSP430 microcontroller with 2 kilobytes of RAM and 60 kilobytes of ROM running at 1 MHz. The microcontroller has the ability to selectively reprogram parts of the on-chip flash memory.

The contributions of this paper are twofold. Our first contribution is that we show the feasibility of loadable programs and services even in a constrained sensor device. The possibility to dynamically load individual programs leads to a very flexible architecture, which still is compact enough for resource constrained sensor nodes. Our second contribution is more general in that we show that preemptive multi-threading does not have to be implemented at the lowest level of the kernel but that it can be built as an application library on top of an event-driven kernel. This allows for thread-based programs running on top of an event-based kernel, without the overhead of reentrancy or multiple stacks in all parts of the system.

8.1.1 Downloading code at run-time

Wireless sensor networks are envisioned to be large scale, with hundreds or even thousands of nodes per network. When developing software for such a large sensor network, being able to dynamically download program code into the network is of great importance. Furthermore, bugs may have to be patched in an operational network [9]. In general, it is not feasible to physically collect and reprogram all sensor devices and in-situ mechanisms are required. A number of methods for distributing code in wireless sensor networks have been developed [21, 8, 17]. For such methods it is important to reduce the number of bytes sent over the network, as communication requires a large parts of the available node energy.

Most operating systems for embedded systems require that a complete binary image of the entire system is built and downloaded into each device. The binary includes the operating system, system libraries, and the actual applications running on top of the system. In contrast, Contiki has the ability to load and unload individual applications or services at run-time. In most cases, an individual application is much smaller than the entire system binary and therefore requires less energy when transmitted through a network. Additionally, the transfer time of an application binary is less than that of an entire system image.

8.1.2 Portability

As the number of different sensor device platforms increases (e.g. [1, 2, 5]), it is desirable to have a common software infrastructure that is portable across hardware platforms. The currently available sensor platforms carry completely different sets of sensors and communication devices. Due to the application specific nature of sensor networks, we do not expect that this will change in the future. The single unifying characteristic of today's platforms is the CPU architecture which uses a memory model without segmentation or memory protection mechanisms. Program code is stored in reprogrammable ROM and data in RAM. We have designed Contiki so that the only abstraction provided by the base system is CPU multiplexing and support for loadable programs and services. As a consequence of the application specific nature of sensor networks, we believe that other abstractions are better implemented as libraries or services and provide mechanisms for dynamic service management.

8.1.3 Event-driven systems

In severely memory constrained environments, a multi-threaded model of operation often consumes large parts of the memory resources. Each thread must have its own stack and because it in general is hard to know in advance how much stack space a thread needs, the stack typically has to be over provisioned. Furthermore, the memory for each stack must be allocated when the thread is created. The memory contained in a stack can not be shared between many concurrent threads, but can only be used by the thread to which is was allocated. Moreover, a threaded concurrency model requires locking mechanisms to prevent concurrent threads from modifying shared resources.

To provide concurrency without the need for per-thread stacks or locking mechanisms, event-driven systems have been proposed [15]. In event-driven systems, processes are implemented as event handlers that run to completion. Because an event handler cannot block, all processes can use the same stack, effectively sharing the scarce memory resources between all processes. Also, locking mechanisms are generally not needed because two event handlers never run concurrently with respect to each other.

While event-driven system designs have been found to work well for many kinds of sensor network applications [18] they are not without problems. The state driven programming model can be hard to manage for programmers [17]. Also, not all programs are easily expressed as state machines. One example is the lengthy computation required for cryptographic operations. Typically, such operations take several seconds to complete on CPU constrained platforms [22]. In a purely event-driven operating system a lengthy computation completely monopolizes the CPU, making the system unable to respond to external events. If the operating system instead was based on preemptive multi-threading this would not be a problem as a lengthy computation could be preempted.

To combine the benefits of both event-driven systems and preemptible threads, Contiki uses a hybrid model: the system is based on an event-driven kernel where preemptive multi-threading is implemented as an application library that is *optionally* linked with programs that *explicitly* require it.

The rest of this paper is structured as follows. Section 8.2 reviews related work and Section 8.3 presents an overview of the Contiki system. We describe the design of the Contiki kernel in Section 8.4. The Contiki service concept is presented in Section 8.5. In the following section, we describe how Contiki handles libraries and communication support is discussed in Section 8.7. We present the implementation of preemptive multi-threading in Section 8.8 and

our experiences with using the system is discussed in Section 8.9. Finally, the paper is concluded in Section 8.10.

8.2 Related work

TinyOS [15] is probably the earliest operating system that directly targets the specific applications and limitations of sensor devices. TinyOS is also built around a lightweight event scheduler where all program execution is performed in tasks that run to completion. TinyOS uses a special description language for composing a system of smaller components [12] which are statically linked with the kernel to a complete image of the system. After linking, modifying the system is not possible [17]. In contrast, Contiki provides a dynamic structure which allows programs and drivers to be replaced during run-time and without relinking.

In order to provide run-time reprogramming for TinyOS, Levis and Culler have developed Maté [17], a virtual machine for TinyOS devices. Code for the virtual machine can be downloaded into the system at run-time. The virtual machine is specifically designed for the needs of typical sensor network applications. Similarly, the MagnetOS [7] system uses a virtual Java machine to distribute applications across the sensor network. The advantages of using a virtual machine instead of native machine code is that the virtual machine code can be made smaller, thus reducing the energy consumption of transporting the code over the network. One of the drawbacks is the increased energy spent in interpreting the code—for long running programs the energy saved during the transport of the binary code is instead spent in the overhead of executing the code. Contiki programs use native code and can therefore be used for all types of programs, including low level device drivers without loss of execution efficiency.

SensorWare [8] provides an abstract scripting language for programming sensors, but their target platforms are not as resource constrained as ours. Similarly, the EmStar environment [13] is designed for less resource constrained systems. Reijers and Langendoen [21] use a patch language to modify parts of the binary image of a running system. This works well for networks where all nodes run the exact same binary code but soon gets complicated if sensors run slightly different programs or different versions of the same software.

The Mantis system [3] uses a traditional preemptive multi-threaded model of operation. Mantis enables reprogramming of both the entire operating system and parts of the program memory by downloading a program image onto

EEPROM, from where it can be burned into flash ROM. Due to the multi-threaded semantics, every Mantis program must have stack space allocated from the system heap, and locking mechanisms must be used to achieve mutual exclusion of shared variables. In contrast, Contiki uses an event based scheduler without preemption, thus avoiding allocation of multiple stacks and locking mechanisms. Preemptive multi-threading is provided by a library that can be linked with programs that explicitly require it.

The preemptive multi-threading in Contiki is similar to fibers [4] and the lightweight fibers approach by Welsh and Mainland [23]. Unlike the lightweight fibers, Contiki does not limit the number of concurrent threads to two. Furthermore, unlike fibers, threads in Contiki support preemption.

As Exokernel [11] and Nemesis [16], Contiki tries to reduce the number of abstractions that the kernel provides to a minimum [10]. Abstractions are instead provided by libraries that have nearly full access to the underlying hardware. While Exokernel strived for performance and Nemesis aimed at quality of service, the purpose of the Contiki design is to reduce size and complexity, as well as to preserve flexibility. Unlike Exokernel, Contiki do not support any protection mechanisms since the hardware for which Contiki is designed do not support memory protection.

8.3 System overview

A running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. A process may be either an application program or a *service*. A service implements functionality used by more than one application process. All processes, both application programs and services, can be dynamically replaced at run-time. Communication between processes always goes through the kernel. The kernel does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware.

A process is defined by an event handler function and an optional poll handler function. The process state is held in the process' private memory and the kernel only keeps a pointer to the process state. On the ESB platform [5], the process state consists of 23 bytes. All processes share the same address space and do not run in different protection domains. Interprocess communication is done by posting events.

A Contiki system is partitioned into two parts: the *core* and the *loaded programs* as shown in Figure 8.1. The partitioning is made at compile time

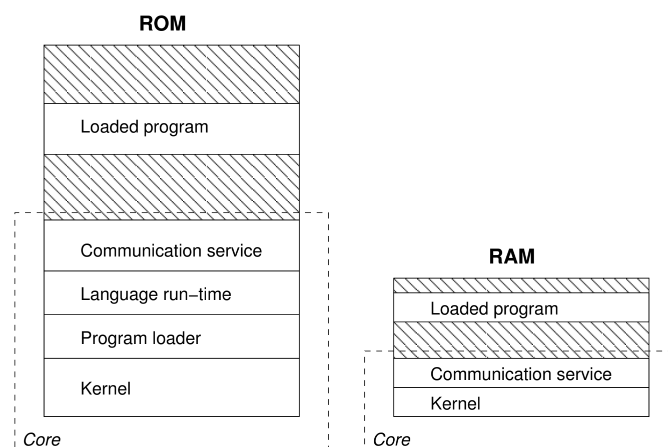

Figure 8.1: Partitioning into core and loaded programs.

and is specific to the deployment in which Contiki is used. Typically, the core consists of the Contiki kernel, the program loader, the most commonly used parts of the language run-time and support libraries, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image that is stored in the devices prior to deployment. The core is generally not modified after deployment, even though it should be noted that it is possible to use a special boot loader to overwrite or patch the core.

Programs are loaded into the system by the program loader. The program loader may obtain the program binaries either by using the communication stack or by using directly attached storage such as EEPROM. Typically, programs to be loaded into the system are first stored in EEPROM before they are programmed into the code memory.

8.4 Kernel architecture

The Contiki kernel consists of a lightweight event scheduler that dispatches events to running processes and periodically calls processes' polling handlers. All program execution is triggered either by events dispatched by the kernel or through the polling mechanism. The kernel does not preempt an event handler once it has been scheduled. Therefore, event handlers must run to completion.

As shown in Section 8.8, however, event handlers may use internal mechanisms to achieve preemption.

The kernel supports two kind of events: *asynchronous* and *synchronous* events. Asynchronous events are a form of deferred procedure call: asynchronous events are enqueued by the kernel and are dispatched to the target process some time later. Synchronous events are similar to asynchronous but immediately causes the target process to be scheduled. Control returns to the posting process only after the target has finished processing the event. This can be seen as an inter-process procedure call and is similar to the door abstraction used in the Spring operating system [14].

In addition to the events, the kernel provides a *polling* mechanism. Polling can be seen as high priority events that are scheduled in-between each asynchronous event. Polling is used by processes that operate near the hardware to check for status updates of hardware devices. When a poll is scheduled all processes that implement a poll handler are called, in order of their priority.

The Contiki kernel uses a single shared stack for all process execution. The use of asynchronous events reduce stack space requirements as the stack is rewound between each invocation of event handlers.

8.4.1 Two level scheduling hierarchy

All event scheduling in Contiki is done at a single level and events cannot preempt each other. Events can only be preempted by interrupts. Normally, interrupts are implemented using hardware interrupts but may also be implemented using an underlying real-time executive. The latter technique has previously been used to provide real-time guarantees for the Linux kernel [6].

In order to be able to support an underlying real-time executive, Contiki never disables interrupts. Because of this, Contiki does not allow events to be posted by interrupt handlers as that would lead to race-conditions in the event handler. Instead, the kernel provides a polling flag that it used to request a poll event. The flag provides interrupt handlers with a way to request immediate polling.

8.4.2 Loadable programs

Loadable programs are implemented using a run-time relocation function and a binary format that contains relocation information. When a program is loaded into the system, the loader first tries to allocate sufficient memory space based

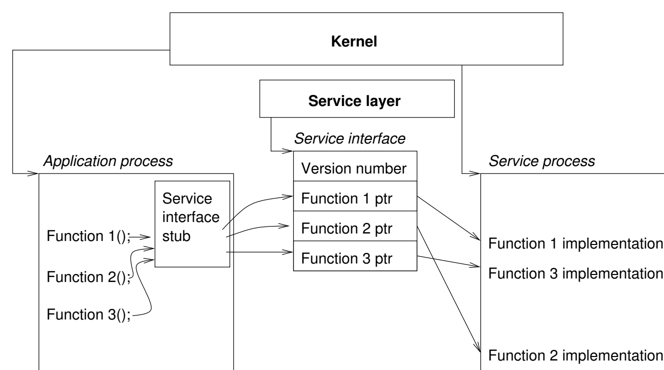

Figure 8.2: An application function calling a service.

on information provided by the binary. If memory allocation fails, program loading is aborted.

After the program is loaded into memory, the loader calls the program's initialization function. The initialization function may start or replace one or more processes.

8.4.3 Power save mode

In sensor networks, being able to power down the node when the network is inactive is an often required way to reduce energy consumption. Power conservation mechanisms depend on both the applications [18] and the network protocols [20]. The Contiki kernel contains no explicit power save abstractions, but lets the the application specific parts of the system implement such mechanisms. To help the application decide when to power down the system, the event scheduler exposes the size of the event queue. This information can be used to power down the processor when there are no events scheduled. When the processor wakes up in response to an interrupt, the poll handlers are run to handle the external event.

8.5 Services

In Contiki, a *service* is a process that implements functionality that can be used by other processes. A service can be seen as a form of a shared library. Services can be dynamically replaced at run-time and must therefore be dynamically linked. Typical examples of services includes communication protocol stacks, sensor device drivers, and higher level functionality such as sensor data handling algorithms.

Services are managed by a *service layer* conceptually situated directly next to the kernel. The service layer keeps track of running services and provides a way to find installed services. A service is identified by a textual string that describes the service. The service layer uses ordinary string matching to querying installed services.

A service consists of a *service interface* and a process that implements the interface. The service interface consists of a version number and a function table with pointers to the functions that implement the interface.

Application programs using the service use a stub library to communicate with the service. The stub library is linked with the application and uses the service layer to find the service process. Once a service has been located, the service stub caches the process ID of the service process and uses this ID for all future requests.

Programs call services through the service interface stub and need not be aware of the fact that a particular function is implemented as a service. The first time the service is called, the service interface stub performs a service lookup in the service layer. If the specified service exists in the system, the lookup returns a pointer to the service interface. The version number in the service interface is checked with the version of the interface stub. In addition to the version number, the service interface contains pointers to the implementation of all service functions. The function implementations are contained in the service process. If the version number of the service stub match the number in the service interface, the interface stub calls the implementation of the requested function.

8.5.1 Service replacement

Like all processes, services may be dynamically loaded and replaced in a running Contiki system. Because the process ID of the service process is used as a service identifier, it is crucial that the process ID is retained if the service process is replaced. For this reason, the kernel provides special mechanism for

replacing a process and retaining the process ID.

When a service is to be replaced, the kernel informs the running version of the service by posting a special event to the service process. In response to this event, the service must remove itself from the system.

Many services have an internal state that may need to be transfered to the new process. The kernel provides a way to pass a pointer to the new service process, and the service can produce a state description that is passed to the new process. The memory for holding the state must be allocated from a shared source, since the process memory is deallocated when the old process is removed.

The service state description is tagged with the version number of the service, so that an incompatible version of the same service will not try to load the service description.

8.6 Libraries

The Contiki kernel only provides the most basic CPU multiplexing and event handling features. The rest of the system is implemented as system libraries that are optionally linked with programs. Programs can be linked with libraries in three different ways. First, programs can be statically linked with libraries that are part of the core. Second, programs can be statically linked with libraries that are part of the loadable program. Third, programs can call services implementing a specific library. Libraries that are implemented as services can be dynamically replaced at run-time.

Typically, run-time libraries such as often-used parts of the language run-time libraries are best placed in the Contiki core. Rarely used or application specific libraries, however, are more appropriately linked with loadable programs. Libraries that are part of the core are always present in the system and do not have to be included in loadable program binaries.

As an example, consider a program that uses the `memcpy()` and `atoi()` functions to copy memory and to convert strings to integers, respectively. The `memcpy()` function is a frequently used C library function, whereas `atoi()` is used less often. Therefore, in this particular example, `memcpy()` has been included in the system core but not `atoi()`. When the program is linked to produce a binary, the `memcpy()` function will be linked against its static address in the core. The object code for the part of the C library that implements the `atoi()` function must, however, be included in the program binary.

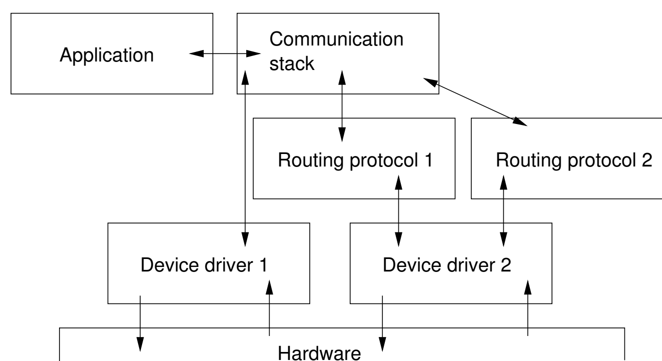

Figure 8.3: Loosely coupled communication stack.

8.7 Communication support

Communication is a fundamental concept in sensor networks. In Contiki, communication is implemented as a service in order to enable run-time replacement. Implementing communication as a service also provides for multiple communication stacks to be loaded simultaneously. In experimental research, this can be used to evaluate and compare different communication protocols. Furthermore, the communication stack may be split into different services as shown in Figure 8.3. This enables run-time replacement of individual parts of the communication stack.

Communication services use the service mechanism to call each other and synchronous events to communicate with application programs. Because synchronous event handlers are required to be run to completion, it is possible to use a single buffer for all communication processing. With this approach, no data copying has to be performed. A device driver reads an incoming packet into the communication buffer and then calls the upper layer communication service using the service mechanisms. The communication stack processes the headers of the packet and posts a synchronous event to the application program for which the packet was destined. The application program acts on the packet contents and optionally puts a reply in the buffer before it returns control to the communication stack. The communication stack prepends its headers to the outgoing packet and returns control to the device driver so that the packet can be transmitted.

mt_yield();
Yield from the running thread.

mt_post(id, event, dataptr);
Post an event from the running thread.

mt_wait(event, dataptr);
Wait for an event to be posted to the running thread.

mt_exit();
Exit the running thread.

mt_start(thread, functionptr, dataptr);
Start a thread with a specified function call.

mt_exec(thread);
Execute the specified thread until it yields or is preempted.

Figure 8.4: The multi-threading library API.

8.8 Preemptive multi-threading

In Contiki, preemptive multi-threading is implemented as a library on top of the event-based kernel. The library is optionally linked with applications that explicitly require a multi-threaded model of operation. The library is divided into two parts: a platform independent part that interfaces to the event kernel, and a platform specific part implementing the stack switching and preemption primitives. Usually, the preemption is implemented using a timer interrupt that saves the processor registers onto the stack and switches back to the kernel stack. In practice very little code needs to be rewritten when porting the platform specific part of the library. For reference, the implementation for the MSP430 consists of 25 lines of C code.

Unlike normal Contiki processes each thread requires a separate stack. The library provides the necessary stack management functions. Threads execute on their own stack until they either explicitly yield or are preempted.

The API of the multi-threading library is shown in Figure 8.4. It consists of four functions that can be called from a running thread (`mt_yield()`, `mt_post()`, `mt_wait()`, and `mt_exit()`) and two functions that are called to setup and run a thread (`mt_start()` and `mt_exec()`). The `mt_exec()`

function performs the actual scheduling of a thread and is called from an event handler.

8.9 Discussion

We have used the Contiki operating system to implement a number of sensor network applications such as multi-hop routing, motion detection with distributed sensor data logging and replication, and presence detection and notification.

8.9.1 Over-the-air programming

We have implemented a simple protocol for over-the-air programming of entire networks of sensors. The protocol transmits a single program binary to selected concentrator nodes using point-to-point communication. The binary is stored in EEPROM and when the entire program has been received, it is broadcasted to neighboring nodes. Packet loss is signaled by neighbors using negative acknowledgments. Repairs are made by the concentrator node. We intend to implement better protocols, such as the Trickle algorithm [19], in the future.

During the development of one network application, a 40-node dynamic distributed alarm system, we used both over-the-air reprogramming and manual wired reprogramming of the sensor nodes. At first, the program loading mechanism was not fully functional and we could not use it during our development. The object code size of our application was approximately 6 kilobytes. Together with the Contiki core and the C library, the complete system image was nearly 30 kilobytes. Reprogramming of an individual sensor node took just over 30 seconds. With 40 nodes, reprogramming the entire network required at least 30 minutes of work and was therefore not feasible to do often. In contrast, over-the-air reprogramming of a single component of the application was done in about two minutes—a reduction in an order of magnitude—and could be done with the sensor nodes placed in the actual test environment.

8.9.2 Code size

An operating system for constrained devices must be compact in terms of both code size and RAM usage in order to leave room for applications running on

Module	**Code size (AVR)**	**Code size (MSP430)**	**RAM usage**
Kernel	1044	810	10 + + $4e$ + $2p$
Service layer	128	110	0
Program loader	-	658	8
Multi-threading	678	582	8 + s
Timer library	90	60	0
Replicator stub	182	98	4
Replicator	1752	1558	200
Total	3874	3876	230 + $4e$ + + $2p$ + s

Table 8.1: Size of the compiled code, in bytes.

top of the system. Table 8.1 shows the compiled code size and the RAM usage of the Contiki system compiled for two architectures: the Texas Instruments MSP430 and the Atmel AVR. The numbers report the size of both core components and an example application: a sensor data replicator service. The replicator service consists of the service interface stub for the service as well as the implementation of the service itself. The program loader is currently only implemented on the MSP430 platform.

The code size of Contiki is larger than that of TinyOS [15], but smaller than that of the Mantis system [3]. Contiki's event kernel is significantly larger than that of TinyOS because of the different services provided. While the TinyOS event kernel only provides a FIFO event queue scheduler, the Contiki kernel supports both FIFO events and poll handlers with priorities. Furthermore, the flexibility in Contiki requires more run-time code than for a system like TinyOS, where compile time optimization can be done to a larger extent.

The RAM requirement depends on the maximum number of processes that the system is configured to have (p), the maximum size of the asynchronous event queue (e) and, in the case of multi-threaded operation, the size of the thread stacks (s).

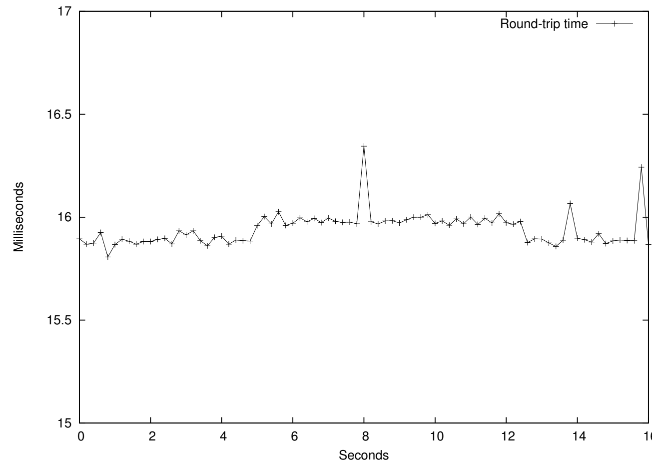

Figure 8.5: A slight increase in response time during a preemptible computation.

8.9.3 Preemption

The purpose of preemption is to facilitate long running computations while being able to react on incoming events such as sensor input or incoming communication packets. Figure 8.5 shows how Contiki responds to incoming packets during an 8 second computation running in a preemptible thread. The curve is the measured round-trip time of 200 "ping" packets of 40 bytes each. The computation starts after approximately 5 seconds and runs until 13 seconds have passed. During the computation, the round-trip time increases slightly but the system is still able to produce replies to the ping packets.

The packets are sent over a 57600 kbit/s serial line with a spacing of 200 ms from a 1.4 GHz PC to an ESB node running Contiki. The packets are transmitted over a serial line rather than over the wireless link in order to avoid radio effects such as bit errors and MAC collisions. The computation consists of an arbitrarily chosen sequence of multiplications and additions that are repeated for about 8 seconds. The cause for the increase in round-trip time during the computation is the cost of preempting the computation and restoring the kernel context before the incoming packet can be handled. The jitter and the spikes of about 0.3 milliseconds seen in the curve can be contributed to activity in other

poll handlers, mostly the radio packet driver.

8.9.4 Portability

We have ported Contiki to a number of architectures, including the Texas Instruments MSP430 and the Atmel AVR. Others have ported the system to the Hitachi SH3 and the Zilog Z80. The porting process consists of writing the boot up code, device drivers, the architecture specific parts of the program loader, and the stack switching code of the multi-threading library. The kernel and the service layer does not require any changes.

Since the kernel and service layer does not require any changes, an operational port can be tested after the first I/O device driver has been written. The Atmel AVR port was made by ourselves in a couple of hours, with help of publicly available device drivers. The Zilog Z80 port was made by a third party, in a single day.

8.10 Conclusions

We have presented the Contiki operating system, designed for memory constrained systems. In order to reduce the size of the system, Contiki is based on an event-driven kernel. The state-machine driven programming of event-driven systems can be hard to use and has problems with handling long running computations. Contiki provides preemptive multi-threading as an application library that runs on top of the event-driven kernel. The library is *optionally* linked with applications that *explicitly* require a multi-threaded model of computation.

A running Contiki system is divided into two parts: a core and loaded programs. The core consists of the kernel, a set of base services, and parts of the language run-time and support libraries. The loaded programs can be loading and unloading individually, at run-time. Shared functionality is implemented as *services*, a form of shared libraries. Services can be updated or replaced individually, which leads to a very flexible structure.

We have shown that dynamic loading and unloading of programs and services is feasible in a resource constrained system, while keeping the base system lightweight and compact. Even though our kernel is event-based, preemptive multi-threading can be provided at the application layer on a per-process basis.

Because of its dynamic nature, Contiki can be used to multiplex the hardware of a sensor network across multiple applications or even multiple users. This does, however, require ways to control access to the reprogramming facilities. We plan to continue our work in the direction of operating system support for secure code updates.

Bibliography

[1] Crossbow mica motes. Web page.
URL: http://www.xbow.com/

[2] Eyes prototype sensor node. Web page. Visited 2004-06-22.
URL: http://eyes.eu.org/sensnet.htm

[3] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: system support for MultimodAl NeTworks of In-Situ sensors. In *Proc. WSNA'03*, 2003.

[4] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proc. USENIX*, 2002.

[5] CST Group at FU Berlin. Scatterweb Embedded Sensor Board. Web page. 2003-10-21.
URL: http://www.scatterweb.com/

[6] M. Barabanov. A Linux-based RealTime Operating System. Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[7] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(2), 2002.

[8] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. MOBISYS'03*, May 2003.

[9] D. Estrin (editor). *Embedded everywhere: A research agenda for networked systems of embedded computers*. National Academy Press, 2001.

[10] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proc. HotOS-V*, May 1995.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc SOSP '95*, December 1995.

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. SIGPLAN'03*, 2003.

[13] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks. In *Proc. USENIX*, 2004.

[14] G. Hamilton and P. Kougiouris. The spring nucleus: A microkernel for objects. In *Proc. Usenix Summer Conf.*, 1993.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. ASPLOS-IX*, November 2000.

[16] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, 14(7):1280–1297, 1996.

[17] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. ASPLOS-X*, October 2002.

[18] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proc. NSDI*, 2004.

[19] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. NSDI*, 2004.

[20] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy aware wireless microsensor networks. *IEEE Signal Processing Magazine*, 19(2):40–50, 2002.

[21] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. WSNA'03*, 2003.

[22] F. Stajano. *Security for Ubiquitous Computing*. Wiley, 2002.

[23] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proc. NSDI*, 2004.

Chapter 9

Paper C: Using Protothreads for Sensor Node Programming

Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. Using protothreads for sensor node programming. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN 2005)*, Stockholm, Sweden, June 2005.

©2005 Swedish Institute of Computer Science.

Abstract

Wireless sensor networks consist of tiny devices that usually have severe resource constraints in terms of energy, processing power and memory. In order to work efficiently within the constrained memory, many operating systems for such devices are based on an event-driven model rather than on traditional multi-threading. While event-driven systems allow for reduced memory usage, they require programs to be developed as explicit state machines. Since implementing programs using explicit state machines is hard, developing and maintaining programs for event-driven systems is typically more difficult than for multi-threaded ones.

In this paper, we introduce protothreads, a programming abstraction for event-driven sensor network systems. Protothreads simplify implementation of high-level functionality on top of event-driven systems, compared to traditional methods.

9.1 Introduction

Wireless sensor networks consist of tiny devices that usually have severe resource constraints in terms of energy, processing power and memory. Most programming environments for wireless sensor network nodes today are based on an event-triggered programming model rather than traditional multi-threading. In TinyOS [7], the event-triggered model was chosen over a multi-threaded model due to the memory overhead of threads. According to Hill et al. [7]:

> "In TinyOS, we have chosen an event model so that high levels of concurrency can be handled in a very small amount of space. A stack-based threaded approach would require that stack space be reserved for each execution context."

While the event-driven model and the threaded model can be shown to be equivalent [9], programs written in the two models typically display differing characteristics [1]. The advantages and disadvantages of the two models are a debated topic [11, 14].

In event-triggered systems, programs are implemented as *event handlers*. Event handlers are invoked in response to external or internal events, and run to completion. An event handler typically is a programming language procedure or function that performs an explicit return to the caller. Because of the run-to-completion semantics, an event-handler cannot execute a *blocking wait*. With run-to-completion semantics, the system can utilize a single, shared stack. This reduces the memory overhead over a multi-threaded system, where memory must be allocated for a stack for each running program.

The run-to-completion semantics of event-triggered systems makes implementing certain high-level operations a complex task. When an operation cannot complete immediately, the operation must be split across multiple invocations of the event handler. Levis et al. [10] refer to this as a split-phase operation. In the words of Levis et al.:

> "This approach is natural for reactive processing and for interfacing with hardware, but complicates sequencing high-level operations, as a logically blocking sequence must be written in a state-machine style."

In this paper, we introduce the notion of using *protothreads* [3, 6] as a method to reduce the complexity of high-level programs in event-triggered sensor node systems. We argue that protothreads can reduce the number of explicit

state machines required to implement typical high-level sensor node programs. We believe this reduction leads to programs that are easier to develop, debug, and maintain, based on extensive experience with developing software for the event-driven uIP TCP/IP stack [4] and Contiki operating system [5].

The main contribution of this paper is the protothread programming abstraction. We show that protothreads reduce the complexity of programming sensor nodes. Further, we demonstrate that protothreads can be implemented in the C programming language, using only standard C language constructs and without any architecture-specific machine code.

The rest of this paper is structured as follows. Section 9.2 presents a motivating example and Section 9.3 introduces the notion of protothreads. Section 9.4 discusses related work, and the paper is concluded in Section 9.5.

9.2 Motivation

To illustrate how high-level functionality is implemented using state machines, we consider a hypothetical energy-conservation mechanism for wireless sensor nodes. The mechanism switches the radio on and off at regular intervals. The mechanism works as follows:

1. Turn radio on.
2. Wait for t_{awake} milliseconds.
3. Turn radio off, but only if all communication has completed.
4. If communication has not completed, wait until it has completed. Then turn off the radio.
5. Wait for t_{sleep} milliseconds. If the radio could not be turned off before t_{sleep} milliseconds because of remaining communication, do not turn the radio off at all.
6. Repeat from step 1.

To implement this protocol in an event-driven model, we first need to identify a set of states around which the state machine can be designed. For this protocol, we can see three states: *on* – the radio is turned on, *waiting* – waiting for any remaining communication to complete, and *off* – the radio is off. Figure 9.2 shows the resulting state machine, including the state transitions.

```
enum {
  ON,
  WAITING,
  OFF
} state;

void radio_wake_eventhandler() {
  switch(state) {

  case OFF:
    if(timer_expired(&timer)) {
      radio_on();
      state = ON;
      timer_set(&timer, T_AWAKE);
    }
    break;

  case ON:
    if(timer_expired(&timer)) {
      timer_set(&timer, T_SLEEP);
      if(!communication_complete())
      {
        state = WAITING;
      } else {
        radio_off();
        state = OFF;
      }
    }
    break;

  case WAITING:
    if(communication_complete()
       || timer_expired(&timer)) {
      state = ON;
      timer_set(&timer, T_AWAKE);
    } else {
      radio_off();
      state = OFF;
    }
    break;
  }
}
```

```
PT_THREAD(radio_wake_thread
          (struct pt *pt)) {
  PT_BEGIN(pt);

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);
    PT_WAIT_UNTIL(pt,
       timer_expired(&timer));

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
      PT_WAIT_UNTIL(pt,
         communication_complete()
         || timer_expired(&timer));
    }

    if(!timer_expired(&timer)) {
      radio_off();
      PT_WAIT_UNTIL(pt,
         timer_expired(&timer));
    }
  }

  PT_END(pt);
}
```

Figure 9.1: The radio sleep cycle implemented with events (left) and with protothreads (right).

To implement this state machine in C, we use an explicit state variable, `state`, that can take on the values OFF, ON, and WAITING. We use a `switch()` statement to perform different actions depending on the `state` variable. The code is placed in an event handler function that is called whenever an event occurs. Possible events in this case are that a timer expires and that communication completes. The resulting C code is shown in the left part of Figure 9.1.

We note that this simple mechanism results in a fairly large amount of C code. The structure of the mechanism, as it is described by the six steps above, is not evident from the C code.

9.3 Protothreads

Protothreads [6] are an extremely lightweight stackless type of threads, designed for severely memory constrained systems. Protothreads provide *conditional blocking waits* on top of an event-driven system, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without complex state machines or full multi-threading.

We developed protothreads in order to deal with the complexity of explicit state machines in the event-driven uIP TCP/IP stack [4]. For uIP, we were able to substantially reduce the number of state machines and explicit states used in the implementations of a number of application level communication protocols. For example, the uIP FTP client could be simplified by completely removing the explicit state machine, and thereby reducing the number of explicit states from 20 to one.

9.3.1 Protothreads versus events

Programs written for an event-driven model typically have to be implemented as explicit state machines. To illustrate how protothreads solve this problem, we return to the radio sleep cycle example from the previous section.

The right part of Figure 9.1 shows how the radio sleep cycle mechanism is implemented with protothreads. Comparing the left and right part of Figure 9.1, we see that the protothreads-based implementation not only is shorter, but also more closely follows the specification of the radio sleep mechanism. Due to the linear code flow of this implementation, the overall logic of the sleep cycle mechanism is visible in the C code. Also, in the protothreads-based implementation we are able to make use of regular C control flow mechanisms such as while() loops and if() statements.

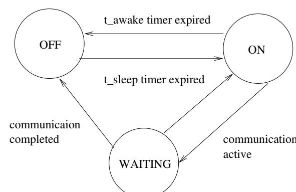

Figure 9.2: State machine realization of the radio sleep cycle protocol.

9.3.2 Protothreads versus threads

The main advantage of protothreads over traditional threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a stack for a thread might use a large part of the available memory. In comparison, a protothread requires only two bytes of memory per protothread and no additional stack.

A protothread runs within a single C function and cannot span over other functions. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls is instead implemented by spawning a separate protothread for each potentially blocking function. Unlike threads, this makes protothreads blocking explicit: the programmer knows exactly which functions that potentially may yield.

9.3.3 Comparison

Table 9.1 summarizes the features of protothreads and compares them with the features of events and threads. The names of the features are from [1].

Feature	Events	Threads	**Proto-threads**
Control structures	No	Yes	**Yes**
Debug stack retained	No	Yes	**Yes**
Implicit locking	Yes	No	**Yes**
Preemption	No	Yes	**No**
Automatic variables	No	Yes	**No**

Table 9.1: Qualitative comparison between events, threads and protothreads

Control structures. One of the advantages of threads over events is that threads allow programs to make full use of the control structures (e.g., *if* conditionals and *while* loops) provided by the programming language. In the event-driven model, control structures must be broken down into two or more pieces in order to implement continuations [1]. Like threads, protothreads allow blocking statements to be used together with control structures.

```
void radio_wake_thread(struct pt *pt) {
  switch(pt->lc) {
  case 0:

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);

    pt->lc = 8;
  case 8:
    if(!timer_expired(&timer)) {
      return;
    }

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {

      pt->lc = 13;
  case 13:
      if(!(communication_complete() ||
           timer_expired(&timer))) {
        return;
      }

    }

    if(!timer_expired(&timer)) {
      radio_off();

      pt->lc = 18;
  case 18:
      if(!timer_expired(&timer)) {
        return;
      }

    }
  }
  }
}
```

Figure 9.3: C switch statement expansion of the protothreads code in Figure 9.1

Debug stack retained. Because the manual stack management and the free flow of control in the event-driven model, debugging is difficult as the sequence of calls is not saved on the stack [1]. With both threads and protothreads, the full call stack is available for debugging.

Implicit locking. With manual stack management, as in the event-driven model, all yield points are immediately visible in the code. This makes it evident to the programmer whether or not a structure needs to be locked. In the threaded model, it is not as evident that a particular function call yields. Using protothreads, however, potentially blocking statements are explicitly implemented with a `PT_WAIT` statement. Program code between such statements never yields.

Preemption. The semantics of the threaded model allows for preemption of a running thread, save its stack, and continue execution of another thread.

Because both the event-driven model and protothreads use a single stack, preemption is not possible within these models.

Automatic variables. Since the threaded model allocates a stack for each thread, automatic variables—variables with function local scope automatically allocated on the stack—are retained even when the thread blocks. Both the event-driven model and protothreads use a single shared stack for all active programs, and rewind the stack every time a program blocks. Therefore, with protothreads, automatic variables are not saved across a blocking wait. This is discussed in more detail below.

9.3.4 Limitations

While protothreads allow programs to take advantage of some of the benefits of a threaded programming model, protothreads also impose some of the limitation from the event-driven model. The most evident limitation from the event-driven model is that automatic variables—variables with function-local scope that are automatically allocated on the stack—are not saved across a blocking wait. While automatic variables can still be used inside a protothread, the contents of the variables must be explicitly stored before executing a wait statement. The reason for this is that protothreads rewind the stack at every blocking statement, and therefore potentially destroy the contents of variables on the stack.

If an automatic variable is erroneously used after a blocking statement, the C compiler is able to detect the problem. Typically a warning is produced, stating that the variable in question "might be used uninitialized in this function". While it may not be immediately apparent for the programmer that this warning is related to the use of automatic variables across a blocking protothreads statement, it does provide an indication that there is a problem with the program. Also, the warning indicates the line number of the problem which assists the programmer in identifying the problem.

The limitation on the use of automatic variables can be handled by using an explicit *state object*, much in the same way as is done in the event-driven model. The state object is a chunk of memory that holds the contents of all automatic variables that need to be saved across a blocking statement. It is, however, the responsibility of the programmer to allocate and maintain such a state object.

It should also be noted that protothreads do not limit the use of *static local* variables. Static local variables are variables that are local in scope but

allocated in the data section. Since these are not placed on the stack, they are not affected by the use of blocking protothreads statements. For functions that do not need to be re-entrant, using static local variables instead of automatic variables can be an acceptable solution to the problem.

9.3.5 Implementation

Protothreads are based on a low-level mechanism that we call *local continuations* [6]. A local continuation is similar to ordinary continuations [12], but does not capture the program stack. Local continuations can be implemented in a variety of ways, including using architecture specific machine code, C-compiler extensions, and a non-obvious use of the C *switch* statement. In this paper, we concentrate on the method based on the C switch statement.

A local continuation supports two operations; it can be either *set* or *resumed*. When a local continuation is set, the state of the function—all CPU registers including the program counter but excluding the stack—is captured. When the same local continuation is resumed, the state of the function is reset to what it was when the local continuation was set.

A protothread consists of a single local continuation. The protothread's local continuation is *set* before each conditional blocking wait. If the condition is true and the wait is to be performed, the protothread executes an explicit return statement, thus returning to the caller. The next time the protothread is called, it *resumes* the local continuation that was previously set. This will effectively cause the program to jump to the conditional blocking wait statement. The condition is re-evaluated and, once the condition is false, the protothread continues down through the function.

```
#define RESUME(lc) switch(lc) { case 0:

#define SET(lc)    lc = __LINE__; case __LINE__:
```

Figure 9.4: The local continuation *resume* and *set* operations implemented using the C switch statement.

Local continuations can be implemented using standard C language constructs and a non-obvious use of the switch statement. With this technique, the local continuation is represented by an unsigned integer. The resume operation is implemented as an open switch statement, and the set operation is

implemented as an assignment of the local continuation and a case statement, as shown in Figure 9.4. Each set operation sets the local continuation to a value that is unique within each function, and the resume operation's switch statement jumps to the corresponding case statement. We note that the case 0: statement in the implementation of the resume operation ensures that the resume statement does nothing if is the local continuation is zero.

Figure 9.3 shows the example radio sleep cycle mechanism from Section 9.2 with the protothreads statements expanded using the C switch implementation of local continuations. We see how each `PT_WAIT` statement has been replaced with a case statement, and how the `PT_BEGIN` statement has been replaced with a switch statement. Additionally, the `PT_END` statement has been replaced with a single right curly bracket, which closes the switch block that was opened by the `PT_BEGIN` statement.

The non-obviousness of the C switch implementation of local continuations is that the technique appears to cause problems when a conditional blocking statement is used inside a nested C control statement. For example, the case 13: statement in Figure 9.3 appears inside an if block, while the corresponding switch statement is located at a higher block. However, this is a valid use of the C switch statement: case statements may be located anywhere inside a switch block. They do not need to be in the same level of nesting, but can be located anywhere, even inside nested if or for blocks. This use of the switch statement is likely to first have been publicly described by Duff [2]. The same technique has later been used by Tatham to implement coroutines in C [13].

The implementation of protothreads using the C switch statements imposes a restriction on programs using protothreads: programs cannot utilize switch statements together with protothreads. If a switch statement is used by the program using protothreads, the C compiler will is some cases emit an error, but in most cases the error not be detected. This is troublesome as it may lead to unexpected run-time behavior which is hard to trace back to an erroneous mixture of one particular implementation of protothreads and switch statements. We have not yet found a suitable solution for this problem.

9.4 Related Work

Kasten and Römer [8] have also identified the need for new abstractions for managing the complexity of event-triggered programming. They introduce OSM, a state machine programming model based on Harel's StateCharts. The model reduces both the complexity of the implementations and the memory

usage. Their work is different from protothreads in that OSM requires support from an external OSM compiler to produce the resulting C code, whereas protothreads only make use of the regular C preprocessor.

9.5 Conclusions

Many operating systems for wireless sensor network nodes are based on an event-triggered programming model. In order to implement high-level operations under this model, programs have to be written as explicit state machines. Software implemented using explicit state machines is often hard to understand, debug, and maintain.

We have presented *protothreads* as a programming abstraction that reduces the complexity of implementations of high-level functionality for event-triggered systems. With protothreads, programs can perform *blocking waits* on top of event-triggered systems with run-to-completion semantics.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European Commission under contract IST-004536-RUNES.

Bibliography

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, pages 289–302, 2002.

[2] T. Duff. Re: Explanation please! Usenet news article, Message-ID: <8144@alice.UUCP>, August 1988.

[3] A. Dunkels. Protothreads web site. Web page. Visited 2006-12-15.
URL: http://www.sics.se/~adam/pt/

[4] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, San Francisco, California, May 2003.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE Emnets '04)*, Tampa, Florida, USA, November 2004.

[6] A. Dunkels and O. Schmidt. Protothreads – Lightweight Stackless Threads in C. Technical Report T2005:05, Swedish Institute of Computer Science.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusets, USA, November 2000.

[8] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, Los Angeles, USA, April 2005.

[9] H. C. Lauer and R. M. Needham. On the duality of operating systems structures. In *Proc. Second International Symposium on Operating Systems*, October 1978.

[10] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'04)*, San Francisco, California, USA, March 2004.

[11] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference, 1996.

[12] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3):233–247, 1993.

[13] S. Tatham. Coroutines in C. Web page, 2000.
URL: http://www.chiark.greenend.org.uk/˜sgtatham/coroutines.html

[14] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue (Kauai), Hawaii, USA, May 2003.

Chapter 10

Paper D: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (ACM SenSys 2006)*, Boulder, Colorado, USA, November 2006.

©2006 Association for Computing Machinery.

Abstract

Event-driven programming is a popular model for writing programs for tiny embedded systems and sensor network nodes. While event-driven programming can keep the memory overhead down, it enforces a state machine programming style which makes many programs difficult to write, maintain, and debug. We present a novel programming abstraction called protothreads that makes it possible to write event-driven programs in a thread-like style, with a memory overhead of only two bytes per protothread. We show that protothreads significantly reduce the complexity of a number of widely used programs previously written with event-driven state machines. For the examined programs the majority of the state machines could be entirely removed. In the other cases the number of states and transitions was drastically decreased. With protothreads the number of lines of code was reduced by one third. The execution time overhead of protothreads is on the order of a few processor cycles.

10.1 Introduction

Event-driven programming is a common programming model for memory-constrained embedded systems, including sensor networks. Compared to multi-threaded systems, event-driven systems do not need to allocate memory for per-thread stacks, which leads to lower memory requirements. For this reason, many operating systems for sensor networks, including TinyOS [19], SOS [17], and Contiki [12] are based on an event-driven model. According to Hill et al. [19]: *"In TinyOS, we have chosen an event model so that high levels of concurrency can be handled in a very small amount of space. A stack-based threaded approach would require that stack space be reserved for each execution context."* Event-driven programming is also often used in systems that are too memory-constrained to fit a general-purpose embedded operating system [28].

An event-driven model does not support a blocking wait abstraction. Therefore, programmers of such systems frequently need to use state machines to implement control flow for high-level logic that cannot be expressed as a single event handler. Unlike state machines that are part of a system specification, the control-flow state machines typically have no formal specification, but are created on-the-fly by the programmer. Experience has shown that the need for explicit state machines to manage control flow makes event-driven programming difficult [3, 25, 26, 35]. With the words of Levis et al. [26]: *"This approach is natural for reactive processing and for interfacing with hardware, but complicates sequencing high-level operations, as a logically blocking sequence must be written in a state-machine style."* In addition, popular programming languages for tiny embedded systems such as the C programming language and nesC [15] do not provide any tools to help the programmer manage the implementation of explicit state machines.

In this paper we study how *protothreads*, a novel programming abstraction that provides a conditional blocking wait operation, can be used to reduce the number of explicit state machines in event-driven programs for memory-constrained embedded systems.

The contribution of this paper is that we show that protothreads simplify event-driven programming by reducing the need for explicit state machines. We show that the protothreads mechanism is simple enough that a prototype implementation of the protothreads mechanism can be done using only C language constructs, without any architecture-specific machine code. We have previously presented the ideas behind protothreads in a position paper [13]. In this paper we significantly extend our previous work by refining the pro-

tothreads mechanism as well as quantifying and evaluating the utility of protothreads.

To evaluate protothreads, we analyze a number of widely used event-driven programs by rewriting them using protothreads. We use three metrics to quantify the effect of protothreads: the number of explicit state machines, the number of explicit state transitions, and lines of code. Our measurements show that protothreads reduce all three metrics for all the rewritten programs. For most programs the explicit state machines can be entirely removed. For the other programs protothreads significantly reduce the number of states. Compared to a state machine, the memory overhead of protothreads is a single byte. The memory overhead of protothreads is significantly lower than for traditional multi-threading. The execution time overhead of protothreads over a state machine is a few processor cycles.

We do not advocate protothreads as a general replacement for state machines. State machines are a powerful tool for designing, modeling, and analyzing embedded systems. They provide a well-founded formalism that allows reasoning about systems and in some cases can provide proofs of the behavior of the system. There are, however, many cases where protothreads can greatly simplify the program without introducing any appreciable memory overhead. Specifically, we have seen many programs for event-driven systems that are based on informally specified state machines. The state machines for those programs are in many cases only visible in the program code and are difficult to extract from the code.

We originally developed protothreads for managing the complexity of explicit state machines in the event-driven uIP embedded TCP/IP stack [10]. The prototype implementations of protothreads presented in this paper are also used in the Contiki operating system [12] and have been used by at least ten different third-party embedded developers for a range of different embedded devices. Examples include an MPEG decoding module for Internet TV-boxes, wireless sensors, and embedded devices collecting data from charge-coupled devices. The implementations have also been ported by others to C++ [30] and Objective C [23].

The rest of the paper is structured as follows. Section 10.2 describes protothreads and shows a motivating example. In Section 10.3 we discuss the memory requirements of protothreads. Section 10.4 shows how state machines can be replaced with protothreads. Section 10.5 describes how protothreads are implemented and presents a prototype implementation in the C programming language. In Section 10.6 we evaluate protothreads, followed by a discussion in Section 10.7. We review of related work in Section 10.8. Finally, the paper

is concluded in Section 10.9.

10.2 Protothreads

Protothreads are a novel programming abstraction that provides a conditional blocking wait statement, PT_WAIT_UNTIL(), that is intended to simplify event-driven programming for memory-constrained embedded systems. The operation takes a conditional statement and blocks the protothread until the statement evaluates to true. If the conditional statement is true the first time the protothread reaches the PT_WAIT_UNTIL() the protothread continues to execute without interruption. The PT_WAIT_UNTIL() condition is evaluated each time the protothread is invoked. The PT_WAIT_UNTIL() condition can be any conditional statement, including complex Boolean expressions.

A protothread is stackless: it does not have a history of function invocations. Instead, all protothreads in a system run on the same stack, which is rewound every time a protothread blocks.

A protothread is driven by repeated calls to the function in which the protothread runs. Because they are stackless, protothreads can only block at the top level of the function. This means that it is not possible for a regular function called from a protothread to block inside the called function - only explicit PT_WAIT_UNTIL() statements can block. The advantage of this is that the programmer always is aware of which statements that potentially may block. Nevertheless, it is possible to perform nested blocking by using hierarchical protothreads as described in Section 10.2.5.

The beginning and the end of a protothread are declared with PT_BEGIN and PT_END statements. Protothread statements, such as the PT_WAIT_UNTIL() statement, must be placed between the PT_BEGIN and PT_END statements. A protothread can exit prematurely with a PT_EXIT statement. Statements outside of the PT_BEGIN and PT_END statements are not part of the protothread and the behavior of such statements are undefined.

Protothreads can be seen as a combination of events and threads. From threads, protothreads have inherited the blocking wait semantics. From events, protothreads have inherited the stacklessness and the low memory overhead. The blocking wait semantics allow linear sequencing of statements in event-driven programs. The main advantage of protothreads over traditional threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained

systems, where a thread's stack might use a large part of the available memory. For example, a thread with a 200 byte stack running on an MS430F149 microcontroller uses almost 10% of the entire RAM. In contrast, the memory overhead of a protothread is as low as two bytes per protothread and no additional stack is needed.

10.2.1 Scheduling

The protothreads mechanism does not specify any specific method to invoke or schedule a protothread; this is defined by the system using protothreads. If a protothread is run on top of an underlying event-driven system, the protothread is scheduled whenever the event handler containing the protothread is invoked by the event scheduler. For example, application programs running on top of the event-driven uIP TCP/IP stack are invoked both when a TCP/IP event occurs and when the application is periodically polled by the TCP/IP stack. If the application program is implemented as a protothread, this protothread is scheduled every time uIP calls the application program.

In the Contiki operating system, processes are implemented as protothreads running on top of the event-driven Contiki kernel. A process' protothread is invoked whenever the process receives an event. The event may be a message from another process, a timer event, a notification of sensor input, or any other type of event in the system. Processes may wait for incoming events using the protothread conditional blocking statements.

The protothreads mechanism does not specify how memory for holding the state of a protothread is managed. As with the scheduling, the system using protothreads decides how memory should be allocated. If the system will run a predetermined amount of protothreads, memory for the state of all protothreads can be statically allocated in advance. Memory for the state of a protothread can also be dynamically allocated if the number of protothreads is not known in advance. In Contiki, the memory for the state of a process' protothread is held in the process control block. Typically, a Contiki program statically allocates memory for its process control blocks.

In general, protothreads are reentrant. Multiple protothreads can be running the same piece of code as long as each protothread has its own memory for keeping state.

```
state: {ON, WAITING, OFF}

radio_wake_eventhandler:
  if (state = ON)
    if (expired(timer))
      timer ← t_sleep
      if (not communication_complete())
        state ← WAITING
        wait_timer ← t_wait_max
      else
        radio_off()
        state ← OFF
  elseif (state = WAITING)
    if (communication_complete() or
        expired(wait_timer))
      state ← OFF
      radio_off()
  elseif (state = OFF)
    if (expired(timer))
      radio_on()
      state ← ON
      timer ← t_awake
```

Figure 10.1: The radio sleep cycle implemented with events, in pseudocode.

10.2.2 Protothreads as Blocking Event Handlers

Protothreads can be seen as blocking event handlers in that protothreads can run on top of an existing event-based kernel, without modifications to the underlying event-driven system. Protothreads running on top of an event-driven system can use the PT_WAIT_UNTIL() statement to block conditionally. The underlying event dispatching system does not need to know whether the event handler is a protothread or a regular event handler.

In general, a protothread-based implementation of a program can act as a drop-in replacement a state machine-based implementation without any modifications to the underlying event dispatching system.

10.2.3 Example: Hypothetical MAC Protocol

To illustrate how protothreads can be used to replace state machines for event-driven programming, we consider a hypothetical energy-conserving sensor network MAC protocol. One of the tasks for a sensor network MAC protocol is to

```
radio_wake_protothread:
  PT_BEGIN
  while (true)
    radio_on()
    timer ← t_awake
    PT_WAIT_UNTIL(expired(timer))
    timer ← t_sleep
    if (not communication_complete())
      wait_timer ← t_wait_max
      PT_WAIT_UNTIL(communication_complete() or
                    expired(wait_timer))
    radio_off()
    PT_WAIT_UNTIL(expired(timer))
  PT_END
```

Figure 10.2: The radio sleep cycle implemented with protothreads, in pseudocode.

allow the radio to be turned off as often as possible in order to reduce the overall energy consumption of the device. Many MAC protocols therefore have scheduled sleep cycles when the radio is turned off completely.

The hypothetical MAC protocol used here is similar to the T-MAC protocol [34] and switches the radio on and off at scheduled intervals. The mechanism is depicted in Figure 10.3 and can be specified as follows:

1. Turn radio on.
2. Wait until $t = t_0 + t_{awake}$.

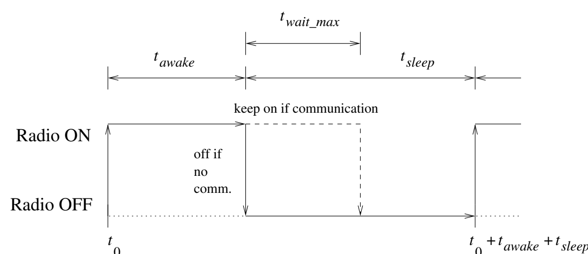

Figure 10.3: Hypothetical sensor network MAC protocol.

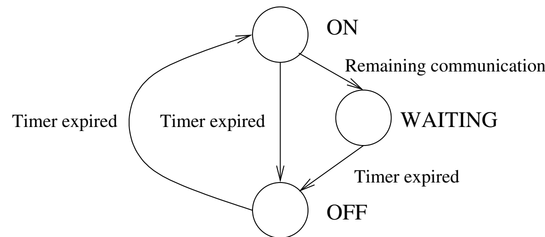

Figure 10.4: State machine realization of the radio sleep cycle of the example MAC protocol.

3. Turn radio off, but only if all communication has completed.
4. If communication has not completed, wait until it has completed or $t = t_0 + t_{awake} + t_{wait_max}$.
5. Turn the radio off. Wait until $t = t_0 + t_{awake} + t_{sleep}$.
6. Repeat from step 1.

To implement this protocol in an event-driven model, we first need to identify a set of states around which the state machine can be designed. For this protocol, we can quickly identify three states: ON – the radio is on, WAITING – waiting for remaining communication to complete, and OFF – the radio is off. Figure 10.4 shows the resulting state machine, including the state transitions.

To implement this state machine, we use an explicit state variable, state, that can take on the values *ON*, *WAITING*, and *OFF*. We use an **if** statement to perform different actions depending on the value of the state variable. The code is placed in an event handler function that is called whenever an event occurs. Possible events in this case are an expiration of a timer and the completion of communication. To simplify the code, we use two separate timers, timer and wait_timer, to keep track of the elapsed time. The resulting pseudocode is shown in Figure 10.1.

We note that this simple mechanism results in a fairly large amount of code. The code that controls the state machine constitutes more than one third of the total lines of code. Also, the six-step structure of the mechanism is not immediately evident from the code.

When implementing the radio sleep cycle mechanism with protothreads we can use the PT_WAIT_UNTIL() statement to wait for the timers to expire.

Figure 10.2 shows the resulting pseudocode code. We see that the code is shorter than the event-driven version from Figure 10.1 and that the code more closely follows the specification of the mechanism.

10.2.4 Yielding Protothreads

Experience with rewriting event-driven state machines to protothreads revealed the importance of an unconditional blocking wait, PT_YIELD(). PT_YIELD() performs an single unconditional blocking wait that temporarily blocks the protothread until the next time the protothread is invoked. At the next invocation the protothread continues executing the code following the PT_YIELD() statement.

With the addition of the PT_YIELD() operation, protothreads are similar to stackless coroutines, much like cooperative multi-threading is similar to stackful coroutines.

10.2.5 Hierarchical Protothreads

While many programs can be readily expressed with a single protothread, more complex operations may need to be decomposed in a hierarchical fashion. Protothreads support this through an operation, PT_SPAWN(), that initializes a child protothread and blocks the current protothread until the child protothread has either ended with PT_END or exited with PT_EXIT. The child protothread is scheduled by the parent protothread; each time the parent protothread is invoked by the underlying system, the child protothread is invoked through the PT_SPAWN() statement. The memory for the state of the child protothread typically is allocated in a local variable of the parent protothread.

As a simple example of how hierarchical protothreads work, we consider a hypothetical data collection protocol that runs in two steps. The protocol first propagates data interest messages through the network. It then continues to propagate data messages back to where the interest messages came from. Both interest messages and data messages are transmitted in a reliable way: messages are retransmitted until an acknowledgment message is received.

Figure 10.5 shows this protocol implemented using hierarchical protothreads. The program consists of a main protothread, data_collection_protocol, that invokes a child protothread, reliable_send, to do transmission of the data.

```
reliable_send(message):
  rxtimer: timer
  PT_BEGIN
  do
    rxtimer ← t_retransmission
    send(message)
    PT_WAIT_UNTIL(ack_received() or expired(rxtimer))
  until (ack_received())
  PT_END

data_collection_protocol
  child_state: protothread_state
  PT_BEGIN
  while (running)
    while (interests_left_to_relay())
      PT_WAIT_UNTIL(interest_message_received())
      send_ack()
      PT_SPAWN(reliable_send(interest), child_state)
    while (data_left_to_relay())
      PT_WAIT_UNTIL(data_message_received())
      send_ack()
      PT_SPAWN(reliable_send(data), child_state)
  PT_END
```

Figure 10.5: Hypothetical data collection protocol implemented with hierarchical protothreads, in pseudocode.

10.2.6 Local Continuations

Local continuations are the low-level mechanism that underpins protothreads. When a protothread blocks, the state of the protothread is stored in a local continuation. A local continuation is similar to ordinary continuations [31] but, unlike a continuation, a local continuation does not capture the program stack. Rather, a local continuation only captures the state of execution inside a single function. The state of execution is defined by the continuation point in the function where the program is currently executing and the values of the function's local variables. The protothreads mechanism only requires that those variables that are actually used across a blocking wait to be stored. However, the current C-based prototype implementations of local continuations depart from this and do not store any local variables.

A local continuation has two operations: *set* and *resume*. When a local continuation is *set*, the state of execution is stored in the local continuation.

This state can then later be restored with the *resume* operation. The state captured by a local continuation does not include the history of functions that have called the function in which the local continuation was *set*. That is, the local continuation does not contain the stack, but only the state of the current function.

A protothread consists of a function and a single local continuation. The protothread's local continuation is *set* before each PT_WAIT_UNTIL() statement. If the condition is false and the wait is to be performed, the protothread is suspended by returning control to the function that invoked the protothread's function. The next time the protothread function is invoked, the protothread *resumes* the local continuation. This effectively causes the program to execute a jump to the conditional blocking wait statement. The condition is reevaluated and either blocks or continues its execution.

10.3 Memory Requirements

Programs written with an event-driven state machine need to store the state of the state machine in a variable in memory. The state can be stored in a single byte unless the state machine has more than 256 states. While the actual program typically stores additional state as program variables, the single byte needed for storing the explicit state constitutes the memory overhead of the state machine. The same program written with protothreads also needs to store the same program variables, and will therefore require exactly the same amount memory as the state machine implementation. The only additional memory overhead is the size of the continuation point. For the prototype C-based implementations, the size of the continuation point is two bytes on the MSP430 and three bytes for the AVR.

In a multi-threading system each thread requires its own stack. Typically, in memory-constrained systems this memory must be statically reserved for the thread and cannot be used for other purposes, even when the thread is not currently executing. Even for systems with dynamic stack memory allocation, thread stacks usually are over-provisioned because of the difficulties of predicting the maximum stack usage of a program, For example, the default stack size for one thread in the Mantis system [2] is 128 bytes, which is a large part of the memory in a system with a few kilobytes of RAM.

In contrast to multi-threading, for event-driven state machines and protothreads all programs run on the same stack. The minimum stack memory requirement is therefore the same as the maximum stack usage of all pro-

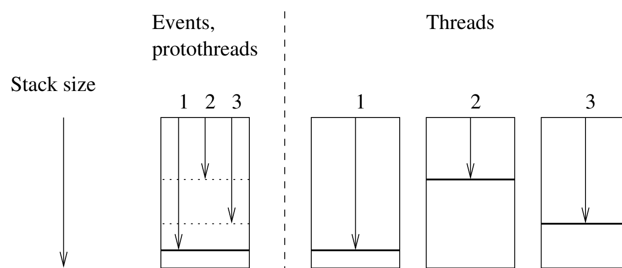

Figure 10.6: The stack memory requirements for three event handlers, the three event handlers rewritten with protothreads, and the equivalent functions running in three threads. Event handlers and protothreads run on the same stack, whereas each thread runs on a stack of its own.

grams. The minimum memory requirement for stacks in a multi-threaded system, however, is the sum of the maximum stack usage of all threads. This is illustrated in Figure 10.6.

10.4 Replacing State Machines with Protothreads

We analyzed a number of existing event-driven programs and found that most control-flow state machines could be decomposed to three primitive patterns: sequences, iterations, and selections. While our findings hold for a number of

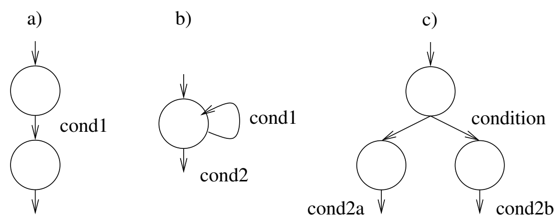

Figure 10.7: Two three primitive state machines: a) sequence, b) iteration, c) selection.

```
a_sequence:
   PT_BEGIN
   (* ... *)
   PT_WAIT_UNTIL(cond1)
   (* ... *)
   PT_END
```

```
an_iteration:
   PT_BEGIN
   (* ... *)
   while (cond1)
      PT_WAIT_UNTIL(cond1 or
                    cond2)
   (* ... *)
   PT_END
```

Figure 10.8: Pseudocode implementation of the sequence and iteration patterns with protothreads.

```
a_selection:
   PT_BEGIN
   (* ... *)
   if (condition)
      PT_WAIT_UNTIL(cond2a)
   else
      PT_WAIT_UNTIL(cond2b)
   (* ... *)
   PT_END
```

Figure 10.9: Pseudocode implementation of the selection pattern with a protothread.

memory-constrained sensor network and embedded programs, our findings are not new in general; Behren et al. [35] found similar results when examining several event-driven systems. Figure 10.7 shows the three primitives. In this section, we show how these state machine primitives map onto protothread constructs and how those can be used to replace state machines.

Figures 10.8 and 10.9 show how to implement the state machine patterns with protothreads. Protothreads allow the programmer to make use of the control structures provided by the programming language: the selection and iteration patterns map onto **if** and **while** statements.

To rewrite an event-driven state machine with protothreads, we first analyse the program to find its state machine. We then map the state machine patterns from Figure 10.7 onto the state machine from the event-driven program. When the state machine patterns have been identified, the program can be rewritten using the code patterns in Figures 10.8 and 10.9.

As an illustration, Figure 10.10 shows the state machine from the radio

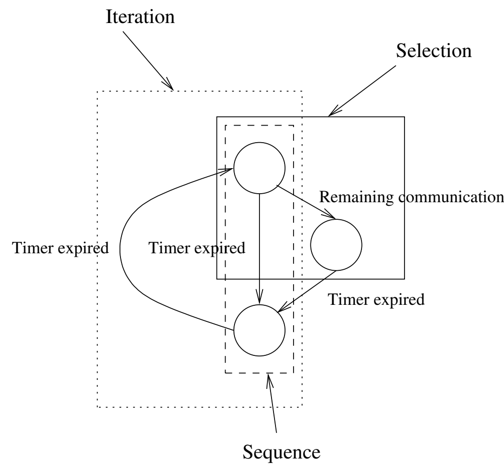

Figure 10.10: The state machine from the example radio sleep cycle mechanism with the iteration and sequence patterns identified.

sleep cycle of the example MAC protocol in Section 10.2.3, with the iteration and sequence state machine patterns identified. From this analysis the protothreads-based code in Figure 10.2 can be written.

10.5 Implementation

We have developed two prototype implementations of protothreads that use only the C preprocessor. The fact that the implementations only depend on the C preprocessor adds the benefit of full portability across all C compilers and of not requiring extra tools in the compilation tool chain. However, the implementations depart from the protothreads mechanism in two important ways: automatic local variables are not saved across a blocking wait statement and C switch and case statements cannot be freely intermixed with protothread-based code. These problems can be solved by implementing protothreads as a special precompiler or by integrating protothreads into existing preprocessor-based languages and C language extensions such as nesC [15].

```
struct pt { lc_t lc };
#define PT_WAITING 0
#define PT_EXITED  1
#define PT_ENDED   2
#define PT_INIT(pt)          LC_INIT(pt->lc)
#define PT_BEGIN(pt)         LC_RESUME(pt->lc)
#define PT_END(pt)           LC_END(pt->lc);    \
                             return PT_ENDED
#define PT_WAIT_UNTIL(pt, c) LC_SET(pt->lc);    \
                             if(!(c))           \
                               return PT_WAITING
#define PT_EXIT(pt)          return PT_EXITED
```

Figure 10.11: C preprocessor implementation of the main protothread operations.

10.5.1 Prototype C Preprocessor Implementations

In the prototype C preprocessor implementation of protothreads the protothread statements are implemented as C preprocessor macros that are shown in Figure 10.11. The protothread operations are a very thin layer of code on top of the local continuation mechanism. The *set* and *resume* operations of the local continuation are implemented as an LC_SET() and the an LC_RESUME() macro. The prototype implementations of LC_SET() and LC_RESUME() depart from the mechanism specified in Section 10.2.6 in that automatic variables are not saved, but only the continuation point of the function.

The PT_BEGIN() statement, which marks the start of a protothread, is implemented with a single LC_RESUME() statement. When a protothread function is invoked, the LC_RESUME() statement will resume the local continuation stored in the protothread's state structure, thus performing an unconditional jump to the last place where the local continuation was set. The resume operation will not perform the jump the first time the protothread function is invoked.

The PT_WAIT_UNTIL() statement is implemented with a LC_SET() operation followed by an `if` statement that performs an explicit `return` if the conditional statement evaluates to false. The returned value lets the caller know that the protothread blocked on a PT_WAIT_UNTIL() statement. PT_END() and PT_EXIT() immediately return to the caller.

To implement yielding protothreads, we need to change the implementa-

```
#define PT_BEGIN(pt) { int yielded = 1;         \
                       LC_RESUME(pt->lc)
#define PT_YIELD(pt)   yielded = 0;             \
                       PT_WAIT_UNTIL(pt, yielded)
#define PT_END(pt)     LC_END(pt->lc);          \
                       return PT_ENDED; }
```

Figure 10.12: Implementation of the PT_YIELD() operation and the updated PT_BEGIN() and PT_END() statements.

```
#define PT_SPAWN(pt, child, thread)            \
        PT_INIT(child);                        \
        PT_WAIT_UNTIL(pt, thread != PT_WAITING)
```

Figure 10.13: Implementation of the PT_SPAWN() operation

tion of PT_BEGIN() and PT_END() in addition to implementing PT_YIELD(). The implementation of PT_YIELD() needs to test whether the protothread has yielded or not. If the protothread has yielded once, then the protothread should continue executing after the PT_YIELD() statement. If the protothread has not yet yielded, it should perform a blocking wait. To implement this, we add an automatic variable, which we call `yielded` for the purpose of this discussion, to the protothread. The `yielded` variable is initialized to one in the PT_BEGIN() statement. This ensures that the variable will be initialized every time the protothread is invoked. In the implementation of PT_YIELD(), we set the variable to zero, and perform a PT_WAIT_UNTIL() that blocks until the variable is non-zero. The next time the protothread is invoked, the conditional statement in the PT_WAIT_UNTIL() is reevaluated. Since the `yielded` variable now has been reinitialized to one, the PT_WAIT_UNTIL() statement will not block. Figure 10.12 shows this implementation of PT_YIELD() and the updated PT_BEGIN() and PT_END() statements.

The implementation of PT_SPAWN(), which is used to implement hierarchical protothreads, is shown in Figure 10.13. It initializes the child protothread and invokes it every time the current protothread is invoked. The PT_WAIT_UNTIL() blocks until the child protothread has exited or ended.

We now discuss how the local continuation functions LC_SET() and LC_RESUME() are implemented.

```
typedef void * lc_t;
#define LC_INIT(c)   c = NULL
#define LC_RESUME(c) if(c) goto *c
#define LC_SET(c)    { __label__ r; r: c = &&r; }
#define LC_END(c)
```

Figure 10.14: Local continuations implemented with the GCC labels-as-values C extension.

GCC C Language Extensions

The widely used GCC C compiler provides a special C language extension that makes the implementation of the local continuation operations straightforward. The C extension, called labels-as-values, makes it possible to save the address of a C label in a pointer. The C goto statement can then be used to jump to the previously captured label. This use of the goto operation is very similar to the unconditional jump most machine code instruction sets provide.

With the labels-as-values C extension, a local continuation simply is a pointer. The *set* operation takes the address of the code executing the operation by creating a C label and capturing its address. The *resume* operation resumes the local continuation with the C goto statement, but only if the local continuation previously has been *set*. The implementation of local continuations with C macros and the labels-as-values C language extension is shown in Figure 10.14. The LC_SET() operation uses the GCC __label__ extension to declare a C label that is local in scope. It then defines the label and stores the address of the label in the local continuation by using the GCC double-ampersand extension.

C Switch Statement

The main problem with the GCC C extension-based implementation of local continuations is that it only works with a single C compiler: GCC. We next show an implementation using only standard ANSI C constructs which uses the C switch statement in a non-obvious way.

Figure 10.15 shows local continuations implemented using the C switch statement. LC_RESUME() is an open switch statement, with a `case 0:` immediately following it. The `case 0:` makes sure that the code after the LC_RESUME() statement is always executed when the local continuation has been initialized with LC_INIT(). The implementation of LC_SET()

```
typedef unsigned short lc_t;
#define LC_INIT(c)   c = 0
#define LC_RESUME(c) switch(c) { case 0:
#define LC_SET(c)    c = __LINE__; case __LINE__:
#define LC_END(c)    }
```

Figure 10.15: Local continuations implemented with the C switch statement.

<table>
<tr>
<td>

```
 1 int sender(pt) {
 2   PT_BEGIN(pt);
 3
 4   /* ... */
 5   do {
 6
 7     PT_WAIT_UNTIL(pt,
 8                cond1);
 9
10   } while(cond);
11   /* ... */
12   PT_END(pt);
13
14 }
```

</td>
<td>

```
int sender(pt) {
  switch(pt->lc) {
  case 0:
    /* ... */
    do {
      pt->lc = 8;
  case 8:
      if(!cond1)
      return PT_WAITING;
    } while(cond);
    /* ... */
  }
  return PT_ENDED;
}
```

</td>
</tr>
</table>

Figure 10.16: Expanded C code with local continuations implemented with the C switch statement.

uses the standard __LINE__ macro. This macro expands to the line number in the source code at which the LC_SET() macro is used. The line number is used as a unique identifier for each LC_SET() statement. The implementation of LC_END() is a single right curly bracket that closes the switch statement opened by LC_RESUME().

To better illustrate how the C switch-based implementation works, Figure 10.16 shows how a short protothreads-based program is expanded by the C preprocessor. We see that the resulting code is fairly similar to how the explicit state machine was implemented in Figure 10.1. However, when looking closer at the expanded C code, we see that the `case 8:` statement on line 7 appears inside the do-while loop, even though the switch statement appears outside of the do-while loop. This does seem surprising at first, but is in fact valid ANSI C code. This use of the switch statement is likely to first have been publicly described by Duff as part of Duff's Device [8]. The same technique has later

been used by Tatham to implement coroutines in C [33].

10.5.2 Memory Overhead

The memory required for storing the state of a protothread, implemented either with the GCC C extension or the C switch statement, is two bytes; the C switch statement-based implementation requires two bytes to store the 16-bit line number identifier of the local continuation. The C extension-based implementation needs to store a pointer to the address of the local continuation. The size of a pointer is processor-dependent but on the MSP430 a pointer is 16 bits, resulting in a two byte memory overhead. A pointer on the AVR is 24 bits, resulting in three bytes of memory overhead. However, the memory overhead is an artifact of the prototype implementations; a precompiler-based implementation would reduce the overhead to one byte.

10.5.3 Limitations of the Prototype Implementations

The two implementations of the local continuation mechanism described above introduce the limitation that automatic variables are not saved across a blocking wait. The C switch-based implementation also limits the use of the C switch statement together with protothread statements.

Automatic Variables

In the C-based prototype implementations, automatic variables—variables with function-local scope that are automatically allocated on the stack—are not saved in the local continuation across a blocking wait. While automatic variables can still be used inside a protothread, the contents of the variables must be explicitly saved before executing a wait statement. Many C compilers, including GCC, detect if automatic local variables are used across a blocking protothreads statement and issues a warning message.

While automatic variables are not preserved across a blocking wait, static local variables are preserved. Static local variables are variables that are local in scope but allocated in the data section of the memory rather than on the stack. Since static local variables are not placed on the stack, they are not affected by the use of blocking protothreads statements. For functions that do not need to be reentrant, static local variables allow the programmer to use local variables inside the protothread.

For reentrant protothreads, the limitation on the use of automatic variables can be handled by using an explicit state object, much in the same way as is commonly done in purely event-driven programs. It is, however, the responsibility of the programmer to allocate and maintain such a state object.

Constraints on Switch Constructs

The implementation of protothreads using the C switch statements imposes a restriction on programs using protothreads: programs cannot utilize switch statements together with protothreads. If a switch statement is used by the program using protothreads, the C compiler will in some cases emit an error, but in most cases the error is not detected by the compiler. This is troublesome as it may lead to unexpected run-time behavior which is hard to trace back to an erroneous mixture of one particular implementation of protothreads and switch statements. We have not yet found a suitable solution for this problem other than using the GCC C extension-based implementation of protothreads.

Possible C Compiler Problems

It could be argued that the use of a non-obvious, though standards-compliant, C construct can cause problems with the C compiler because the nested switch statement may not be properly tested. We have, however, tested protothreads on a wide range of C compilers and have only found one compiler that was not able to correctly parse the nested C construct. In this case, we contacted the vendor who was already aware of the problem and immediately sent us an updated version of the compiler. We have also been in touch with other C compiler vendors, who have all assured us that protothreads work with their product.

10.5.4 Alternative Approaches

In addition to the implementation techniques described above, we examine two alternative implementation approaches: implementation with assembly language and with the C language functions setjmp and longjmp.

Assembly Language

We have found that for some combinations of processors and C compilers it is possible to implement protothreads and local continuations by using assembly language. The *set* of the local continuations is then implemented as a C

function that captures the return address from the stack and stores it in the local continuation, along with any callee save registers. Conversely, the *resume* operation would restore the saved registers from the local continuation and perform an unconditional jump to the address stored in the local continuation. The obvious problem with this approach is that it requires a porting effort for every new processor and C compiler. Also, since both a return address and a set of registers need to be stored in the local continuation, its size grows. However, we found that the largest problem with this approach is that some C compiler optimizations will make the implementation difficult. For example, we were not able to produce a working implementation with this method for the Microsoft Visual C++ compiler.

With C setjmp and longjmp Functions

While it at first seems possible to implement the local continuation operations with the setjmp and longjmp functions from the standard C library, we have seen that such an implementation causes subtle problems. The problem is because the setjmp and longjmp function store and restore the stack pointer, and not only the program counter. This causes problems when the protothread is invoked through different call paths since the stack pointer is different with different call paths. The *resume* operation would not correctly resume a local continuation that was *set* from a different call path.

We first noticed this when using protothreads with the uIP TCP/IP stack. In uIP application protothreads are invoked from different places in the TCP/IP code depending on whether or not a TCP retransmission is to take place.

Stackful Approaches

By letting each protothread run on its own stack it would be possible to implement the full protothread mechanism, including storage of automatic variables across a blocking wait. With such an implementation the stack would be switched to the protothread's own stack by the PT_BEGIN operation and switched back when the protothread blocks or exits. This approach could be implemented with a coroutine library or the multi-threading library of Contiki. However, this implementation would result in a memory overhead similar to that of multi-threading because each invocation of a protothread would require the same amount of stack memory as the equivalent protothread running in a thread of its own due to the stack space required by functions called from within the protothread.

Finally, a promising alternative method is to store a copy the stack frame of the protothread function in the local continuation when the protothread blocks. This saves all automatic variables of the protothread function across a blocking wait, including variables that are not used after the blocking wait. Since all automatic variables are saved, this approach have a higher memory overhead. Furthermore, this approach requires both C compiler-specific and CPU architecture-specific code, thus reducing the portability of the implementation. However, the extra porting effort may be outweighed by the benefits of storing automatic variables across blocking waits. We will continue to pursue this as future work.

10.6 Evaluation

To evaluate protothreads we first measure the reduction in code complexity that protothreads provide by reimplementing a set of event-driven programs with protothreads and measure the complexity of the resulting code. Second, we measure the memory overhead of protothreads compared to the memory overhead of an event-driven state machine. Third, we compare the execution time overhead of protothreads with that of event-driven state machines.

10.6.1 Code Complexity Reduction

To measure the code complexity reduction of protothreads we reimplement parts of a number of event-driven applications with protothreads: XNP [20], the previous default over-the-air programming program from TinyOS; the buffer management module of TinyDB [27], a database engine for TinyOS; radio protocol drivers for the Chipcon CC1000 and RF Monolithics TR1001 radio chips; the SMTP client in the uIP embedded TCP/IP stack and a code propagation program from the Contiki operating system. The state machines in XNP, TinyDB, and the CC1000 drivers were rewritten by applying the method for replacing state machines with protothreads from Section 10.4 whereas the TR1001 driver, the uIP SMTP client and the Contiki code propagation were rewritten from scratch.

We use three metrics to measure the complexity of the programs we reimplemented with protothreads: the number of explicit states, the number of explicit state transitions, as well as the lines of code of the reimplemented functions.

All reimplemented programs consist of complex state machines. Using

protothreads, we were able to entirely remove the explicit state machines for most programs. For all programs, protothreads significantly reduce the number of state transitions and lines of code.

The reimplemented programs have undergone varying amounts of testing. The Contiki code propagation, the TR1001 low-level radio driver, and the uIP SMTP client are well tested and are currently used on a daily basis in live systems, XNP and TinyDB have been verified to be working but not heavily tested, and the CC1000 drivers have been tested and run in simulation.

Furthermore, we have anecdotal evidence to support our hypothesis that protothreads are an alternative to state machines for embedded software development. The protothreads implementations have for some time been available as open source on our web page [9]. We know that at least ten embedded systems developers have successfully used protothreads to replace state machines for embedded software development. Also, our protothreads code have twice been recommended by experienced embedded developers in Jack Ganssle's embedded development newsletter [14].

XNP

XNP [20] is one of the in-network programming protocols used in TinyOS [19]. XNP downloads a new system image to a sensor node and writes the system image to the flash memory of the device. XNP is implemented on top of the event-driven TinyOS. Therefore, any operations in XNP that would be blocking in a threaded system have to be implemented as state machines. We chose XNP because it is a relatively complex program implemented on top of an event-driven system. The implementation of XNP has previously been analyzed by Jeong [20], which assisted us in our analysis. The implementation of XNP consists of a large switch statement with 25 explicit states, encoded as defined constants, and 20 state transitions. To analyze the code, we identified the state transitions from manual inspection of the code inside the switch statement.

Since the XNP state machine is implemented as one large switch statement, we expected it to be a single, complex state machine. But, when drawing the state machine from analysis of the code, it turned out that the switch statement in fact implements five different state machines. The entry points of the state machines are not immediately evident from the code, as the state of the state machine was changed in several places throughout the code.

The state machines we found during the analysis of the XNP program are shown in Figure 10.17. For reasons of presentation, the figure does not show

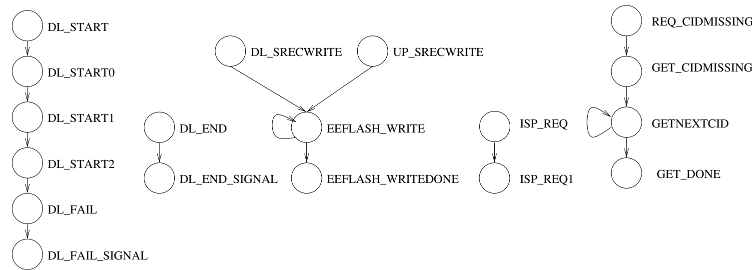

Figure 10.17: XNP state machines. The names of the states are from the code. The IDLE and ACK states are not shown.

the IDLE and ACK states. Almost all states have transitions to one of these states. If an XNP operation completes successfully, the state machine goes into the ACK state to transmit an acknowledgment over the network. The IDLE state is entered if an operation ends with an error, and when the acknowledgment from the ACK state has been transmitted.

In the figure we clearly see many of the state machine patterns from Figure 10.7. In particular, the sequence pattern is evident in all state machines. By using the techniques described in Section 10.4 we were able to rewrite all state machines into protothreads. Each state machine was implemented as its own protothread.

The IDLE and ACK states are handled in a hierarchical protothread. A separate protothread is created for sending the acknowledgment signal. This protothread is spawned from the main protothread every time the program logic dictates that an acknowledgment should be sent.

TinyDB

TinyDB [27] is a small database engine for the TinyOS system. With TinyDB, a user can query a wireless sensor network with a database query language similar to SQL. TinyDB is one of the largest TinyOS programs available.

In TinyOS long-latency operations are split-phase [15]. Split-phase operations consist of two parts: a request and a completion event. The request completes immediately, and the completion event is posted when the operation has completed. TinyDB contains a large number of split-phase operations. Since programs written for TinyOS cannot perform a blocking wait, many complex

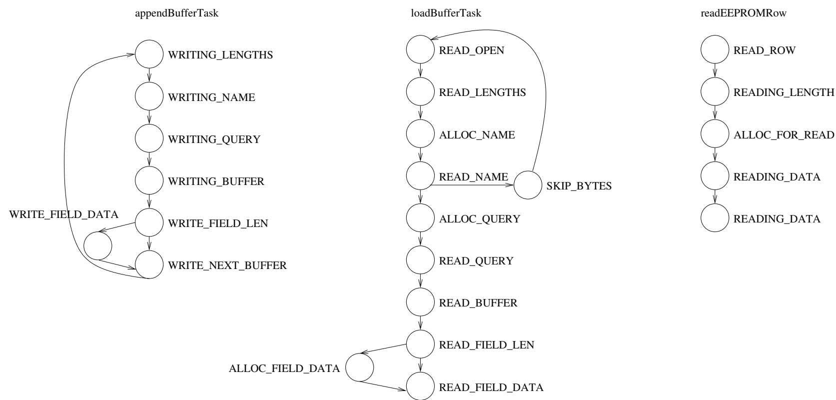

Figure 10.18: Three state machines from TinyDB.

operations in TinyDB are encoded as state machines.

To the state machines in TinyDB we analyze the TinyDB buffer management module, DBBufferC. DBBufferC uses the MemAlloc module to allocate memory. Memory allocation requests are performed from inside a function that drives the state machine. However, when the request is completed, the allocComplete event is handled by a different function. This event handler must handle the event different depending on the state of the state machine. In fact, the event handler itself implements a small piece of the entire state machine. The fact that the implementation of the state machine is distributed across different functions makes the analysis of the state machine difficult.

From inspection of the DBBufferC code we found the three state machines in Figure 10.18. We also found that there are more state machines in the code, but we were not able to adequately trace them because the state transitions were scattered around the code. By rewriting the discovered state machines with protothreads, we were able to completely remove the explicit state machines.

Low Level Radio Protocol Drivers

The Chipcon CC1000 and RF Monolithics TR1001 radio chips are used in many wireless sensor network devices. Both chips provide a very low-level interface to the radio. The chips do not perform any protocol processing themselves but interrupt the CPU for every incoming byte. All protocol functional-

ity, such as packet framing, header parsing, and MAC protocol must be implemented in software.

We analyze and rewrite CC1000 drivers from the Mantis OS [2] and from SOS [17], as well as the TR1001 driver from Contiki [12]. All drivers are implemented as explicit state machines. The state machines run in the interrupt handlers of the radio interrupts.

The CC1000 driver in Mantis has two explicit state machines: one for handling and parsing incoming bytes and one for handling outgoing bytes. In contrast, both the SOS CC1000 driver and the Contiki TR1001 drivers have only one state machine that parses incoming bytes. The state machine that handles transmissions in the SOS CC1000 driver is shown in Figure 10.19. The structures of the SOS CC1000 driver and the Contiki TR1001 driver are very similar.

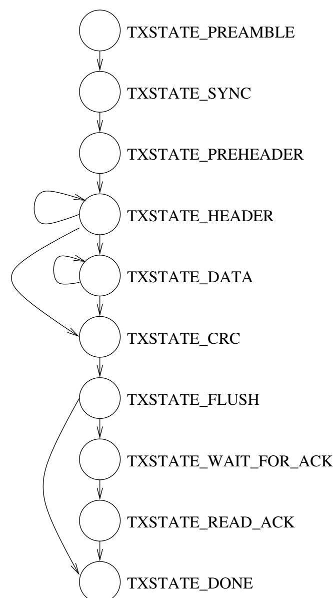

Figure 10.19: Transmission state machine from the SOS CC1000 driver.

With protothreads we could replace most parts of the state machines. However, for both the SOS CC1000 driver and the Contiki TR1001 drivers, we kept a top-level state machine. The reason for this is that those state machines were

not used to implement control flow. The top-level state machine in the SOS CC1000 driver controlled if the driver was currently transmitting or receiving a packet, or if it was finding a synchronization byte.

uIP TCP/IP Stack

The uIP TCP/IP stack [10] is designed for memory-constrained embedded systems and therefore has a very low memory overhead. It is used in embedded devices from well over 30 companies, with applications ranging from pico-satellites to car traffic monitoring systems. To reduce the memory overhead uIP follows the event-driven model. Application programs are implemented as event handlers and to achieve blocking waits, application programs need to be written as explicit state machines. We have rewritten the SMTP client in uIP with protothreads and were able to completely remove the state machines.

Contiki

The Contiki operating system [12] for wireless sensor networks is based on an event-driven kernel, on top of which protothreads provide a thread-like programming style. The first version of Contiki was developed before we introduced protothreads. After developing protothreads, we found that they reduced the complexity of writing software for Contiki.

For the purpose of this paper, we measure the implementation of a distribution program for distributing and receiving binary code modules for the Contiki dynamic loader [11]. The program was initially implemented without protothreads but was later rewritten when protothreads were introduced to Contiki. The program can be in one of three modes: (1) receiving a binary module from a TCP connection and loads it into the system, (2) broadcasting the binary module over the wireless network, and (3) receiving broadcasts of a binary module from a nearby node and loading it into memory.

When rewriting the program with protothreads, we removed most of the explicit state machines, but kept four states. These states keep track in which mode the program is: if it is receiving or broadcasting a binary module.

Results

The results of reimplementing the programs with protothreads are presented in Table 10.1. The lines of code reported in the table are those of the rewritten functions only. We see that in all cases the number of states, state transitions, and lines of code were reduced by rewriting the programs with protothreads. In

Program	States, before	States, after	Transitions, before	Transitions, after
XNP	25	-	20	-
TinyDB	23	-	24	-
Mantis CC1000 driver	15	-	19	-
SOS CC1000 driver	26	9	32	14
Contiki TR1001 driver	12	3	32	3
uIP SMTP client	10	-	10	-
Contiki code propagation	6	4	11	3

Program	Lines of code, before	Lines of code, after	Reduction, percentage
XNP	222	152	32%
TinyDB	374	285	24%
Mantis CC1000 driver	164	127	23%
SOS CC1000 driver	413	348	16%
Contiki TR1001 driver	152	77	49%
uIP SMTP client	223	122	45%
Contiki code propagation	204	144	29%

Table 10.1: The number of explicit states, explicit state transitions, and lines of code before and after rewriting with protothreads.

most cases the rewrite completely removed the state machine. The total average reduction in lines of code is 31%. For the programs rewritten by applying the replacement method from Section 10.4 (XNP, TinyDB, and the CC1000 drivers) the average reduction is 23% and for the programs that were rewritten from scratch (the TR1001 driver, the uIP SMTP client, and the Contiki code propagation program) the average reduction is 41%.

Table 10.2 shows the compiled code size of the rewritten functions when written as a state machine and with protothreads. We see that the code size increases in most cases, except for the Contiki code propagation program. The average increase for the programs where the state machines were replaced with protothreads by applying the method from Section 10.4 is 14%. The Contiki TR1001 driver is only marginally larger when written with protothreads. The uIP SMTP client, on the other hand, is significantly larger when written with protothreads rather than with a state machine. The reason for this is that the code for creating SMTP message strings could be optimized through code reuse in the state machine-based implementation, something which was not

Program	Code size, before (bytes)	Code size, after (bytes)	Increase
XNP	931	1051	13%
TinyDB DBBufferC	2361	2663	13%
Mantis CC1000	994	1170	18%
SOS CC1000	1912	2165	13%
Contiki TR1001	823	836	2%
uIP SMTP	1106	1901	72%
Contiki code prop.	1848	1426	-23%

Table 10.2: Code size before and after rewriting with protothreads.

possible in the protothreads-based implementation without significantly sacrificing readability. In contrast with the uIP SMTP client, the Contiki code propagation program is significantly smaller when written with protothreads. Here, it was possible to optimize the protothreads-based code by code reuse, which was not readily possible in the state machine-based implementation.

We conclude that protothreads reduce the number of states and state transitions and the lines of code, at the price of an increase in code size. The size of the increase, however, depends on the properties of the particular program rewritten with protothreads. It is therefore not possible to draw any general conclusions from measured increase in code size.

10.6.2 Memory Overhead

We measured the stack size required for the Contiki TR1001 driver and the Contiki code propagation mechanism running on the MSP430 microcontroller. We measure the stack usage by filling the stack with a known byte pattern, running the TR1001 driver, and inspecting the stack to see how much of the byte pattern that was overwritten.

Neither the Contiki code propagation mechanism nor the Contiki TR1001 driver use the stack for keeping state in program variables. Therefore the entire stack usage of the two programs when running as threads is overhead.

We compare the memory overhead of the Contiki TR1001 driver and the Contiki code propagation running as threads with the memory overhead of a state machine and protothreads in Table 10.3.

	State machine	Proto-thread	Thread
Contiki TR1001 driver	1	2	18
Contiki code propagation	1	2	34

Table 10.3: Memory overhead in bytes for the Contiki TR1001 driver and the Contiki code propagation on the MSP430, implemented with a state machine, a protothread, and a thread.

	State machine	Proto-thread	Yielding protothread
MSP430	9	12	17
AVR	23	34	45

Table 10.4: Machine code instructions overhead for a state machine, a protothread, and a yielding protothread.

10.6.3 Run-time Overhead

To evaluate the run-time overhead of protothreads we counted the machine code instruction overhead of protothreads, compared to a state machine, for the MSP430 and the AVR microcontrollers. Furthermore, we measured the execution time for the driver for the TR1001 radio chip from Contiki, implemented both as a state machine and as a protothread. We also compare the numbers with measurements on a cooperative multi-threading implementation of said program. We used the cooperative user space multi-threading library from Contiki [12] which is a standard stack-switching multi-threading library.

Machine Code Instruction Overhead

To analyze the number of additional machine code instructions in the execution path for protothreads compared to state machines, we compiled our program using the GCC C compiler version 3.2.3 for the MSP430 microcontroller and GCC version 3.4.3 for the AVR microcontroller.

With manual inspection of the generated object machine code we counted the number of machine code instructions needed for the switch and case statements in a state machine function and for the protothread operations in a protothread function. The results are given in Table 10.4. The absolute overhead

Compiler optimization	State machine (ms)	Protothreads, GCC C extension (ms)	Protothreads, C switch statement (ms)
Size (-Os)	0.0373	0.0397	0.0434
Speed (-O1)	0.0369	0.0383	0.0415

Compiler optimization	State machine (cycles)	Protothreads, GCC C extension (cycles)	Protothreads, C switch statement (cycles)
Size (-Os)	91.67	97.56	106.7
Speed (-O1)	90.69	94.12	102.0

Table 10.5: Mean execution time in milliseconds and processor cycles for a single invocation of the TR1001 input driver under Contiki on the MSP430 platform.

for protothreads over the state machine is very small: three machine code instructions for the MSP430 and 11 for the AVR. In comparison, the number of instructions required to perform a context switch in the Contiki implementation of cooperative multi-threading is 51 for the MSP430 and 80 for the AVR.

The additional instructions for the protothread in Table 10.4 are caused by the extra case statement that is included in the implementation of the PT_BEGIN operation.

Execution Time Overhead

To measure the execution time overhead of protothreads over that of an event-driven state machine, we implemented the low-level radio protocol driver for the RFM TR1001 radio chip from Contiki using both an event-driven state machine and a yielding protothread. We measured the execution time by feeding the driver with data from 1000 valid radio packets and measured the average execution time of the driver's function. The time was measured using a periodic timer interrupt driven by the MSP430 timer A1 at a rate of 1000 Hz. We set the clock speed of the MSP430 digitally controlled oscillator to 2.4576 MHz. For the protothreads-based implementation we measured both the GCC C extension-based and the C switch statement-based implementations of the local continuations operations.

The measurement results are presented in Table 10.5. We see that the average execution time overhead of protothreads is low: only about five cycles

per invocation for the GCC C extension-based implementation and just over ten cycles per invocation for the C switch-based implementation. The results are consistent with the machine code overhead in Table 10.4. We also measured the execution time of the radio driver rewritten with cooperative multi-threading and found it to be approximately three times larger than that of the protothread-based implementation because of the overhead of the stack switching code in the multi-threading library.

Because of the low execution time overhead of protothreads we conclude that protothreads are usable even for interrupt handlers with tight execution time constraints.

10.7 Discussion

We have been using the prototype implementations of protothreads described in this paper in Contiki for two years and have found that the biggest problem with the prototype implementations is that automatic variables are not preserved across a blocking wait. Our workaround is to use static local variables rather than automatic variables inside protothreads. While the use of static local variables may be problematic in the general case, we have found it to work well for Contiki because of the small scale of most Contiki programs. Also, as many Contiki programs do not need to be reentrant, the use of static local variables work well.

Code organization is different for programs written with state machines and with protothreads. State machine-based programs tend to consist either of a single large function containing a large state machine or of many small functions where the state machine is difficult to find. On the contrary, protothreads-based programs tend to be based around a single protothread function that contains the high-level logic of the program. If the underlying event system calls different functions for every incoming event, a protothreads-based program typically consists of a single protothread function and a number of small event handlers that invoke the protothread when an event occurs.

10.8 Related Work

Research in the area of software development for sensor networks has led to a number of new abstractions that aim at simplifying sensor network programming [1, 4]. Approaches with the same goal include virtual machines [25] and macro-programming of sensors [16, 29, 37]. Protothreads differ from these

sensor network programming abstractions in that we target the difficulty of low-level event-driven programming rather than the difficulty of developing application software for sensor networks.

Kasten and Römer [21] have also identified the need for new abstractions for managing the complexity of event-triggered state machine programming. They introduce OSM, a state machine programming model based on Harel's StateCharts[18] and use the Esterel language. The model reduces both the complexity of the implementations and the memory usage. Their work is different from protothreads in that they help programmers manage their state machines, whereas protothreads are designed to reduce the number of state machines. Furthermore, OSM requires support from an external OSM compiler to produce the resulting C code, whereas the prototype implementations of protothreads only make use of the regular C preprocessor.

Simpson [32] describes a cooperative mini-kernel mechanism for C++ which is very similar to our protothreads in that it was designed to replace state machines. Simpson's mechanism also uses a single stack. However, it needs to implement its own stack to track the location of a yield point. In contrast, protothreads do not use a stack, but hold all their state in a 16-bit integer value.

Lauer and Needham [24] proved, essentially, that events and threads are duals of each other and that the same program can be written for either of the two systems. With protothreads, we put this into practice by actually rewriting event-driven programs with blocking wait semantics.

Protothreads are similar to coroutines [22] in the sense that a protothread continues to execute at the point of the last return from the function. In particular, protothreads are similar to asymmetric coroutines [7], just as cooperative multi-threading is similar to asymmetric coroutines. However, unlike coroutines and cooperative multi-threading, protothreads are stackless and can only block at the top level of the protothread function.

Dabek et al. [6] present libasynch, a C++ library that assists the programmer in writing event-driven programs. The library provides garbage collection of allocated memory as well as a type-safe way to pass state between callback functions. However, the libasynch library does not provide sequential execution and software written with the library cannot use language statements to control program flow across blocking calls.

The Capriccio system by von Behren et al. [36] shows that in a memory-rich environment, threads can be made as memory efficient as events. This requires modification to the C compiler so that it performs dynamic memory allocation of stack space during run-time. However, as dynamic memory al-

location quickly causes memory fragmentation in the memory-constrained devices for which the protothreads mechanism is designed, dynamic allocation of stack memory is not feasible.

Cunningham and Kohler [5] develop a library to assist programmers of event-driven systems in program analysis, along with a tool for visualizing callback chains, as well as a tool for verifying properties of programs implemented using the library. Their work is different from ours in that they help the programmer manage the state machines for event-driven programs, whereas protothreads are designed to replace such state machines.

Adya et al. [3] discuss the respective merits of event-driven and threaded programming and present a hybrid approach that shows that the event-driven and multi-threaded code can coexist in a unified concurrency model. The authors have developed a set of adaptor functions that allows event-driven code to call threaded code, and threaded code to call event-driven code, without requiring that the caller has knowledge about the callee's approach.

State machines are a powerful tool for developing real-time systems. Abstractions such as Harel's StateCharts [18] are designed to help developers to develop and manage state machines and are very valuable for systems that are designed as state machines. Protothreads, in contrast, are intended to replace state machines with sequential C code. Also, protothreads do not provide any mechanisms for assisting development of hard real-time systems.

10.9 Conclusions

We present protothreads, a novel abstraction for memory-constrained embedded systems. Due to memory-constraints, such systems are often based on an event-driven model. Experience has shown that event-driven programming is difficult because the lack of a blocking wait abstraction forces programmers to implement control flow with state machines.

Protothreads simplify programming by providing a conditional blocking wait operation, thereby reducing the need for explicit state machines. Protothreads are inexpensive: the memory overhead is only two bytes per protothread.

We develop two prototype protothreads implementations using only C preprocessor and evaluate the usefulness of protothreads by reimplementing some widely used event-driven programs using protothreads. Our results show that for most programs the explicit state machines could be entirely removed. Furthermore, protothreads significantly reduce the number of state transitions and

lines of code. The execution time overhead of protothreads is on the order of a few processor cycles. We find the code size of a program written with protothreads to be slightly larger than the equivalent program written as a state machine.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European Commission under contract IST-004536-RUNES. Thanks go to Kay Römer and Umar Saif for reading and suggesting improvements on drafts of this paper, and to our paper shepherd Philip Levis for his many insightful comments that significantly helped to improve the paper.

Bibliography

[1] T. Abdelzaher, J. Stankovic, S. Son, B. Blum, T. He, A. Wood, and C. Lu. A communication architecture and programming abstractions for real-time embedded sensor networks. In *Workshop on Data Distribution for Real-Time Systems*, Providence, RI, USA, May 2003.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 50–59, San Diego, CA, USA, September 2003.

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, pages 289–302, 2002.

[4] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A programming model for event-driven embedded systems. In *Proc. of the 18th Annual ACM Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA, March 2003.

[5] R. Cunningham and E. Kohler. Making events less slippery with eel. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)*, Santa Fee, New Mexico, June 2005. IEEE Computer Society.

[6] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 2002 SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[7] A. L. de Moura and R. Ierusalimschy. Revisiting coroutines. MCC 15/04, PUC-Rio, Rio de Janeiro, RJ, June 2004.

[8] T. Duff. Unwinding loops. Usenet news article, net.lang.c, Message-ID: <2748@alice.UUCP>, May 1984.

[9] A. Dunkels. Protothreads web site. Web page. Visited 2006-04-06.
URL: http://www.sics.se/~adam/pt/

[10] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, San Francisco, California, May 2003.

[11] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006*, Boulder, Colorado, USA, 2006.

[12] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE Emnets '04)*, Tampa, Florida, USA, November 2004.

[13] A. Dunkels, O. Schmidt, and T. Voigt. Using protothreads for sensor node programming. In *Proc. of the Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, Stockholm, Sweden, June 2005.

[14] J. Ganssle. The embedded muse. Monthly newsletter.
URL: http://www.ganssle.com/tem-back.htm

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, San Diego, California, USA, June 2003.

[16] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *Proc. of Distributed Computing in Sensor Systems (DCOSS)'05*, Marina del Rey, CA, USA, June 2005.

[17] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys '05)*, Seattle, WA, USA, June 2005.

[18] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[19] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusets, USA, November 2000.

[20] J. Jeong. Analysis of xnp network reprogramming module. Web page, October 2003. Visited 2006-04-06.
URL: http://www.cs.berkeley.edu/˜jaein/cs294_1/xnp_anal.htm

[21] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, Los Angeles, USA, April 2005.

[22] D. E. Knuth. *The art of computer programming, volume 1: fundamental algorithms (2nd edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[23] Framework Labs. Protothreads for Objective-C/Cocoa. Visited 2006-04-06.
URL: http://www.frameworklabs.de/protothreads.html

[24] H. C. Lauer and R. M. Needham. On the duality of operating systems structures. In *Proc. Second International Symposium on Operating Systems*, October 1978.

[25] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[26] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'04)*, San Francisco, California, USA, March 2004.

[27] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[28] M. Melkonian. Get by Without an RTOS. *Embedded Systems Programming*, 13(10), September 2000.

[29] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *Proc. IPSN'05*, Los Angeles, CA, USA, April 2005.

[30] J. Paisley and J. Sventek. Real-time detection of grid bulk transfer traffic. In *Proceedings of the 10th IEEE/IFIP Network Operations Management Symposium*, Vancouver, Canada, April 2006.

[31] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3):233–247, 1993.

[32] Z. B. Simpson. State machines: Cooperative mini-kernels with yielding. In *Computer Game Developer's Conference*, Austin, TX, November 1999.

[33] S. Tatham. Coroutines in C. Web page, 2000.
URL: http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

[34] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 171–180, Los Angeles, California, USA, 2003.

[35] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue (Kauai), Hawaii, USA, May 2003.

[36] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proc. SOSP '03*, pages 268–281, 2003.

[37] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'04)*, San Francisco, California, USA, March 2004.

Chapter 11

Paper E: Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks

Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (ACM SenSys 2006)*, Boulder, Colorado, USA, November 2006.

©2006 Association for Computing Machinery.

Abstract

From experience with wireless sensor networks it has become apparent that dynamic reprogramming of the sensor nodes is a useful feature. The resource constraints in terms of energy, memory, and processing power make sensor network reprogramming a challenging task. Many different mechanisms for reprogramming sensor nodes have been developed ranging from full image replacement to virtual machines.

We have implemented an in-situ run-time dynamic linker and loader that use the standard ELF object file format. We show that run-time dynamic linking is an effective method for reprogramming even resource constrained wireless sensor nodes. To evaluate our dynamic linking mechanism we have implemented an application-specific virtual machine and a Java virtual machine and compare the energy cost of the different linking and execution models. We measure the energy consumption and execution time overhead on real hardware to quantify the energy costs for dynamic linking.

Our results suggest that while in general the overhead of a virtual machine is high, a combination of native code and virtual machine code provide good energy efficiency. Dynamic run-time linking can be used to update the native code, even in heterogeneous networks.

11.1 Introduction

Wireless sensor networks consist of a collection of programmable radio-equipped embedded systems. The behavior of a wireless sensor network is encoded in software running on the wireless sensor network nodes. The software in deployed wireless sensor network systems often needs to be changed, both to update the system with new functionality and to correct software bugs. For this reason dynamically reprogramming of wireless sensor network is an important feature. Furthermore, when developing software for wireless sensor networks, being able to update the software of a running sensor network greatly helps to shorten the development time.

The limitations of communication bandwidth, the limited energy of the sensor nodes, the limited sensor node memory which typically is on the order of a few thousand bytes large, the absence of memory mapping hardware, and the limited processing power make reprogramming of sensor network nodes challenging.

Many different methods for reprogramming sensor nodes have been developed, including full system image replacement [14, 16], approaches based on binary differences [15, 17, 31], virtual machines [18, 19, 20], and loadable native code modules in the first versions of Contiki [5] and SOS [12]. These methods are either inefficient in terms of energy or require non-standard data formats and tools.

The primary contribution of this paper is that we investigate the use of standard mechanisms and file formats for reprogramming sensor network nodes. We show that in-situ dynamic run-time linking and loading of native code using the ELF file format, which is a standard feature on many operating systems for PC computers and workstations, is feasible even for resource-constrained sensor nodes. Our secondary contribution is that we measure and quantify the energy costs of dynamic linking and execution of native code and compare it to the energy cost of transmission and execution of code for two virtual machines: an application-specific virtual machine and the Java virtual machine.

We have implemented a dynamic linker in the Contiki operating system that can link, relocate, and load standard ELF object code files. Our mechanism is independent of the particular microprocessor architecture on the sensor nodes and we have ported the linker to two different sensor node platforms with only minor modifications to the architecture dependent module of the code.

To evaluate the energy costs of the dynamic linker we implement an application specific virtual machine for Contiki together with a compiler for a subset of Java. We also adapt the Java virtual machine from the lejOS sys-

tem [8] to run under Contiki. We measure the energy cost of reprogramming and executing a set of program using dynamic linking of native code and the two virtual machines. Using the measurements and a simple energy consumption model we calculate break-even points for the energy consumption of the different mechanisms. Our results suggest that while the execution time overhead of a virtual machine is high, a combination of native code and virtual machine code may give good energy efficiency.

The remainder of this paper is structured as follows. In Section 11.2 we discuss different scenarios in which reprogramming is useful. Section 11.3 presents a set of mechanisms for executing code inside a sensor node and in Section 11.4 we discuss loadable modules and the process of linking, relocating, and loading native code. Section 11.5 describes our implementation of dynamic linking and our virtual machines. Our experiments and the results are presented in Section 11.6 and discuss the results in Section 11.7. Related work is reviewed in Section 11.8. Finally, we conclude the paper in Section 11.9.

11.2 Scenarios for Software Updates

Software updates for sensor networks are necessary for a variety of reasons ranging from implementation and testing of new features of an existing program to complete reprogramming of sensor nodes when installing new applications. In this section we review a set of typical reprogramming scenarios and compare their qualitative properties.

11.2.1 Software Development

Software development is an iterative process where code is written, installed, tested, and debugged in a cyclic fashion. Being able to dynamically reprogram parts of the sensor network system helps shorten the time of the development cycle. During the development cycle developers typically change only one part of the system, possibly only a single algorithm or a function. A sensor network used for software development may therefore see large amounts of small changes to its code.

11.2.2 Sensor Network Testbeds

Sensor network testbeds are an important tool for development and experimentation with sensor network applications. New applications can be tested in a

realistic setting and important measurements can be obtained [36]. When a new application is to be tested in a testbed the application typically is installed in the entire network. The application is then run for a specified time, while measurements are collected both from the sensors on the sensor nodes, and from network traffic.

For testbeds that are powered from a continuous energy source, the energy consumption of software updates is only of secondary importance. Instead, qualitative properties such as ease of use and flexibility of the software update mechanism are more important. Since the time required to make an update is important, the throughput of a network-wide software update is of importance. As the size of the transmitted binaries impact the throughput, the binary size still can be used as an evaluation metric for systems where throughput is more important than energy consumption.

11.2.3 Correction of Software Bugs

The need for correcting software bugs in sensor networks was early identified [7]. Even after careful testing, new bugs can occur in deployed sensor networks caused by, for example, an unexpected combination of inputs or variable link connectivity that stimulate untested control paths in the communication software [30].

Software bugs can occur at any level of the system. To correct bugs it must therefore be possible to reprogram all parts of the system.

11.2.4 Application Reconfiguration

In an already installed sensor network, the application may need to be reconfigured. This includes change of parameters, or small changes in the application such as changing from absolute temperature readings to notification when thresholds are exceeded [26]. Even though reconfiguration not necessarily include software updates [25], application reconfiguration can be done by reprogramming the application software. Hence software updates can be used in an application reconfiguration scenario.

11.2.5 Dynamic Applications

There are many situations where it is useful to replace the application software of an already deployed sensor network. One example is the forest fire detection scenario presented by Fok et al. [9] where a sensor network is used to detect

Scenario	Update frequency	Update fraction	Update level	Program longevity
Development	Often	Small	All	Short
Testbeds	Seldom	Large	All	Long
Bug fixes	Seldom	Small	All	Long
Reconfig.	Seldom	Small	App	Long
Dynamic Application	Often	Small	App	Long

Table 11.1: Qualitative comparison between different reprogramming scenarios.

a fire. When the fire detection application has detected a fire, the fire fighters might want to run a search and rescue application as well as a fire tracking application. While it may possible to host these particular applications on each node despite the limited memory of the sensor nodes, this approach is not scalable [9]. In this scenario, replacing the application on the sensor nodes leads to a more scalable system.

11.2.6 Summary

Table 11.1 compares the different scenarios and their properties. *Update fraction* refers to what amount of the system that needs to be updated for every update, *update level* to at what levels of the system updates are likely to occur, and *program longevity* to how long an installed program will be expected to reside on the sensor node.

11.3 Code Execution Models and Reprogramming

Many different execution models and environments have been developed or adapted to run on wireless sensor nodes. Some with the notion of facilitating programming [1], others motivated by the potential of saving energy costs for reprogramming enabled by the compact code representation of virtual machines [19]. The choice of the execution model directly impacts the data format and size of the data that needs to be transported to a node. In this section we discuss three different mechanisms for executing program code inside each sensor node: script languages, virtual machines, and native code.

11.3.1 Script Languages

There are many examples of script languages for embedded systems, including BASIC variants, Python interpreters [22], and TCL machines [1]. However, most script interpreters target platforms with much more resources than our target platforms and we have therefore not included them in our comparison.

11.3.2 Virtual Machines

Virtual machines are a common approach to reduce the cost of transmitting program code in situations where the cost of distributing a program is high. Typically, program code for a virtual machine can be made more compact than the program code for the physical machine. For this reason virtual machines are often used for programming sensor networks [18, 19, 20, 23].

While many virtual machines such as the Java virtual machine are generic enough to perform well for a variety of different types of programs, most virtual machines for sensor networks are designed to be highly configurable in order to allow the virtual machine to be tailored for specific applications. In effect, this means that parts of the application code is implemented as virtual machine code running on the virtual machine, and other parts of the application code is implemented in native code that can be used from the programs running on the virtual machine.

11.3.3 Native Code

The most straightforward way to execute code on sensor nodes is by running native code that is executed directly by the microcontroller of the sensor node. Installing new native code on a sensor node is more complex than installing code for a virtual machine because the native code uses physical addresses which typically need to be updated before the program can be executed. In this section we discuss two widely used mechanisms for reprogramming sensor nodes that execute native code: full image replacement and approaches based on binary differences.

Full Image Replacement

The most common way to update software in embedded systems and sensor networks is to compile a complete new binary image of the software together

with the operating system and overwrite the existing system image of the sensor node. This is the default method used by the XNP and Deluge network reprogramming software in TinyOS [13].

The full image replacement does not require any additional processing of the loaded system image before it is loaded into the system, since the loaded image resides at the same, known, physical memory address as the previous system image. For some systems, such as the Scatterweb system code [33], the system contains both an operating system image and a small set of functions that provide functionality for loading new operating system images. A new operating system image can overwrite the existing image without overwriting the loading functions. The addresses of the loading functions are hard-coded in the operating system image.

Diff-based Approaches

Often a small update in the code of the system, such as a bugfix, will cause only minor differences between in the new and old system image. Instead of distributing a new full system image the binary differences, deltas, between the modified and original binary can be distributed. This reduces the amount of data that needs to be transferred. Several types of diff-based approaches have been developed [15, 17, 31] and it has been shown that the size of the deltas produced by the diff-based approaches is very small compared to the full binary image.

11.4 Loadable Modules

A less common alternative to full image replacement and diff-based approaches is to use loadable modules to perform reprogramming. With loadable modules, only parts of the system need to be modified when a single program is changed. Typically, loadable modules require support from the operating system. Contiki and SOS are examples of systems that support loadable modules and TinyOS is an example of an operating system without loadable module support.

A loadable module contains the native machine code of the program that is to be loaded into the system. The machine code in the module usually contains references to functions or variables in the system. These references must be resolved to the physical address of the functions or variables before the machine code can be executed. The process of resolving those references is called

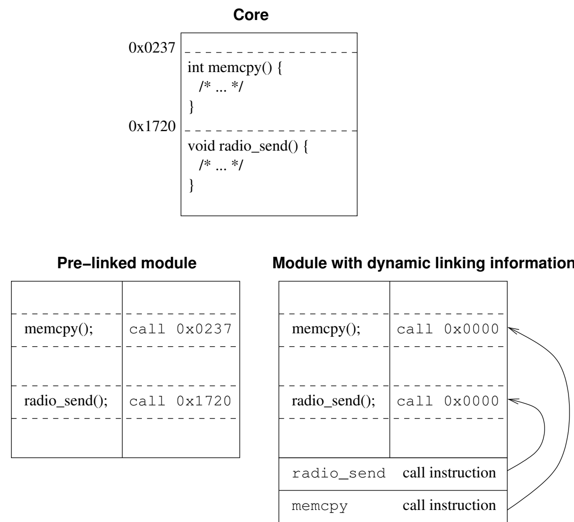

Figure 11.1: The difference between a pre-linked module and a module with dynamic linking information: the pre-linked module contains physical addresses whereas the dynamically linked module contains symbolic names.

linking. Linking can be done either when the module is compiled or when the module is loaded. We call the former approach pre-linking and the latter dynamic linking. A pre-linked module contains the absolute physical addresses of the referenced functions or variables whereas a dynamically linked module contains the symbolic names of all system core functions or variables that are referenced in the module. This information increases the size of the dynamically linked module compared to the pre-linked module. The difference is shown in Figure 11.1. Dynamic linking has not previously been considered for wireless sensor networks because of the perceived run-time overhead, both in terms of execution time, energy consumption, and memory requirements.

The machine code in the module usually contains references not only to functions or variables in the system, but also to functions or variables within the module itself. The physical address of those functions will change depending on the memory address at which the module is loaded in the system. The addresses of the references must therefore be updated to the physical address that the function or variable will have when the module is loaded. The process

of updating these references is known as relocation. Like linking, relocation can be done either at compile-time or at run-time.

When a module has been linked and relocated the program loader loads the module into the system by copying the linked and relocated native code into a place in memory from where the program can be executed.

11.4.1 Pre-linked Modules

The machine code of a pre-linked module contains absolute addresses of all functions and variables in the system code that are referenced by the module. Linking of the module is done at compile time and only relocation is performed at run-time. To link a pre-linked module, information about the physical addresses of all functions and variables in the system into which the module is to be loaded must be available at compile time.

There are two benefits of pre-linked modules over dynamically linked modules. First, pre-linked modules are smaller than dynamically linked modules which results in less information to be transmitted. Second, the process of loading a pre-linked module into the system is less complex than the process of linking a dynamically linked module. However, the fact that all physical addresses of the system core are hard-coded in the pre-linked module is a severe drawback as a pre-linked module can only be loaded into a system with the exact same physical addresses as the system that was to generate the list of addresses that was used for linking the module.

In the original Contiki system [5] we used pre-linked binary modules for dynamic loading. When compiling the Contiki system core, the compiler generated a map file containing the mapping between all globally visible functions and variables in the system core and their addresses. This list of addresses was used to pre-link Contiki modules.

We quickly noticed that while pre-linked binary modules worked well for small projects with a homogeneous set of sensor nodes, the system quickly became unmanageable when the number of sensor nodes grew. Even a small change to the system core of one of the sensor nodes would make it impossible to load binary a module into the system bedcase the addresses of variables and functions in the core were different from when the program was linked. We used version numbers to guard against this situation. Version numbers did help against system crashes, but did not solve the general problem: new modules could not be loaded into the system.

11.4.2 Dynamic Linking

With dynamic linking, the object files do not only contain code and data, but also names of functions are variables of the system core that are referenced by the module. The code in the object file cannot be executed before the physical addresses of the referenced variables and functions have been filled in. This process is done at run time by a dynamic linker.

In the Contiki dynamic linker we use two file formats for the dynamically linked modules, ELF and Compact ELF.

ELF - Executable and Linkable Format

One of the most common object code format for dynamic linking is the Executable and Linkable Format (ELF) [3]. It is a standard format for object files and executables that is used for most modern Unix-like systems. An ELF object file include both program code and data and additional information such as a symbol table, the names of all external unresolved symbols, and relocation tables. The relocation tables are used to locate the program code and data at other places in memory than for which the object code originally was assembled. Additionally, ELF files can hold debugging information such as the line numbers corresponding to specific machine code instructions, and file names of the source files used when producing the ELF object.

ELF is also the default object file format produced by the GCC utilities and for this reason there are a number of standard software utilities for manipulating ELF files available. Examples include debuggers, linkers, converters, and programs for calculating program code and data memory sizes. These utilities exist for a wide variety of platforms, including MS Windows, Linux, Solaris, and FreeBSD. This is a clear advantage over other solutions such as FlexCup [27], which require specialized utilities and tools.

Our dynamic linker in Contiki understands the ELF format and is able to perform dynamic linking, relocation, and loading of ELF object code files. The debugging features of the ELF format are not used.

CELF - Compact ELF

One problem with the ELF format is the overhead in terms of bytes to be transmitted across the network, compared to pre-linked modules. There are a number of reasons for the extra overhead. First, ELF, as any dynamically relocatable file format, includes the symbolic names of all referenced functions or variables that need to be linked at run-time. Second, and more important,

the ELF format is designed to work on 32-bit and 64-bit architectures. This causes all ELF data structures to be defined with 32-bit data types. For 8-bit or 16-bit targets the high 16 bits of these fields are unused.

To quantify the overhead of the ELF format we devise an alternative to the ELF object code format that we call CELF - Compact ELF. A CELF file contains the same information as an ELF file, but represented with 8 and 16-bit datatypes. CELF files typically are half the size of the corresponding ELF file. The Contiki dynamic loader is able to load CELF files and a utility program is used to convert ELF files to CELF files.

It is possible to further compress CELF files using lossless data compression. However, we leave the investigation of the energy-efficiency of this approach to future work.

The drawback of the CELF format is that it requires a special compressor utility is for creating the CELF files. This makes the CELF format less attractive for use in many real-world situations.

11.4.3 Position Independent Code

To avoid performing the relocation step when loading a module, it is in some cases possible to compile the module into position independent code. Position independent code is a type of machine code which does not contain any absolute addresses to itself, but only relative references. This is the approach taken by the SOS system.

To generate position independent code compiler support is needed. Furthermore, not all CPU architectures support position independent code and even when supported, programs compiled to position independent code typically are subject to size restrictions. For example, the AVR microcontroller supports position independent code but restricts the size of programs to 4 kilobytes. For the MSP430 no compiler is known to fully support position independent code.

11.5 Implementation

We have implemented run-time dynamic linking of ELF and CELF files in the Contiki operating system [5]. To evaluate dynamic linking we have implemented an application specific virtual machine for Contiki together with a compiler for a subset of Java, and have ported a Java virtual machine to Contiki.

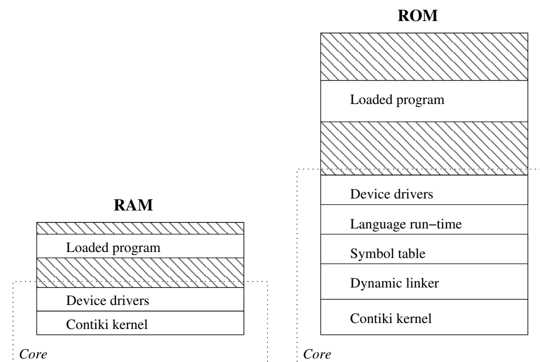

Figure 11.2: Partitioning in Contiki: the core and loadable programs in RAM and ROM.

11.5.1 The Contiki Operating System

The Contiki operating system was the first operating system for memory-constrained sensor nodes to support dynamic run-time loading of native code modules. Contiki is built around an event-driven kernel and has very low memory requirements. Contiki applications run as extremely lightweight protothreads [6] that provide blocking operations on top of the event-driven kernel at a very small memory cost. Contiki is designed to be highly portable and has been ported to over ten different platforms with different CPU architectures and using different C compilers.

A Contiki system is divided into two parts: the core and the loadable programs as shown in Figure 11.2. The core consists of the Contiki kernel, device drivers, a set of standard applications, parts of the C language library, and a symbol table. Loadable programs are loaded on top of the core and do not modify the core.

The core has no information about the loadable programs, except for information that the loadable programs explicitly register with the core. Loadable programs, on the other hand, have full knowledge of the core and may freely call functions and access variables that reside in the core. Loadable programs can call each other by going through the kernel. The kernel dispatches calls from one loaded program to another by looking up the target program in an in-kernel list of active processes. This one-way dependency makes it possible to load and unload programs at run-time without needing to patch the core and

without the need for a reboot when a module has been loaded or unloaded.

While it is possible to replace the core at run-time by running a special loadable program that overwrites the current core and reboots the system, experience has shown that this feature is not often used in practice.

11.5.2 The Symbol Table

The Contiki core contains a table of the symbolic names of all externally visible variable and function names in the Contiki core and their corresponding addresses. The table includes not only the Contiki system, but also the C language run-time library. The symbol table is used by the dynamic linker when linking loaded programs.

The symbol table is created when the Contiki core binary image is compiled. Since the core must contain a correct symbol table, and a correct symbol table cannot be created before the core exists, a three-step process is required to compile a core with a correct symbol table. First, an intermediary core image with an empty symbol table is compiled. From the intermediary core image an intermediary symbol table is created. The intermediary symbol table contains the correct symbols of the final core image, but the addresses of the symbols are incorrect. Second, a second intermediary core image that includes the intermediary symbol table is created. This core image now contains a symbol table of the same size as the one in the final core image so the addresses of all symbols in the core are now as they will be in the final core image. The final symbol table is then created from the second intermediary core image. This symbol table contains both the correct symbols and their correct addresses. Third, the final core image with the correct symbol table is compiled.

The process of creating a core image is automated through a simple `make` script. The symbol table is created using a combination of standard ELF tools.

For a typical Contiki system the symbol table contains around 300 entries which amounts to approximately 4 kilobytes of data stored in flash ROM.

11.5.3 The Dynamic Linker

We implemented a dynamic linker for Contiki that is designed to link, relocate, and load either standard ELF files [3] and CELF, Compact ELF, files. The dynamic linker reads ELF/CELF files through the Contiki virtual filesystem interface, CFS, which makes the dynamic linker unaware of the physical location of the ELF/CELF file. Thus the linker can operate on files stored either in

RAM, on-chip flash ROM, external EEPROM, or external ROM without modification. Since all file access to the ELF/CELF file is made through the CFS, the dynamic linker does not need to concern itself with low-level filesystem details such as wear-leveling or fragmentation [4] as this is better handled by the CFS.

The dynamic linker performs four steps to link, relocate and load an ELF/CELF file. The dynamic linker first parses the ELF/CELF file and extracts relevant information about where in the ELF/CELF file the code, data, symbol table, and relocation entries are stored. Second, memory for the code and data is allocated from flash ROM and RAM, respectively. Third, the code and data segments are linked and relocated to their respective memory locations, and fourth, the code is written to flash ROM and the data to RAM.

Currently, memory allocation for the loaded program is done using a simple block allocation scheme. More sophisticated allocation schemes will be investigated in the future.

Linking and Relocating

The relocation information in an ELF/CELF file consists of a list of relocation entries. Each relocation entry corresponds to an instruction or address in the code or data in the module that needs to be updated with a new address. A relocation entry contains a pointer to a symbol, such as a variable name or a function name, a pointer to a place in the code or data contained in the ELF/CELF file that needs to be updated with the address of the symbol, and a relocation type which specifies how the data or code should be updated. The relocation types are different depending on the CPU architecture. For the MSP430 there is only one single relocation type, whereas the AVR has 19 different relocation types.

The dynamic linker processes a relocation entry at a time. For each relocation entry, its symbol is looked up in the symbol table in the core. If the symbol is found in the core's symbol table, the address of the symbol is used to patch the code or data to which the relocation entry points. The code or data is patched in different ways depending on the relocation type and on the CPU architecture.

If the symbol in the relocation entry was not found in the symbol table of the core, the symbol table of the ELF/CELF file itself is searched. If the symbol is found, the address that the symbol will have when the program has been loaded is calculated, and the code or data is patched in the same way as if the symbol was found in the core symbol table.

Relocation entries may also be relative to the data, BSS, or code segment in the ELF/CELF file. In that case no symbol is associated with the relocation entry. For such entries the dynamic linker calculates the address that the segment will have when the program has been loaded, and uses that address to patch the code or data.

Loading

When the linking and relocating is completed, the text and data have been relocated to their final memory position. The text segment is then written to flash ROM, at the location that was previously allocated. The memory allocated for the data and BSS segments are used as an intermediate storage for transferring text segment data from the ELF/CELF file before it is written to flash ROM. Finally, the memory allocated for the BSS segment is cleared, and the contents of the data segment is copied from the ELF/CELF file.

Executing the Loaded Program

When the dynamic linker has successfully loaded the code and data segments, Contiki starts executing the program.

The loaded program may replace an already running Contiki service. If the service that is to be replaced needs to pass state to the newly loaded service, Contiki supports the allocation of an external memory buffer for this purpose. However, experience has shown that this mechanism has been very scarcely used in practice and the mechanism is likely to be removed in future versions of Contiki.

Portability

Since the ELF/CELF format is the same across different platforms, we designed the Contiki dynamic linker to be easily portable to new platforms. The loader is split into one architecture specific part and one generic part. The generic part parses the ELF/CELF file, finds the relevant sections of the file, looks up symbols from the symbol table, and performs the generic relocation logic. The architecture specific part does only three things: allocates ROM and RAM, writes the linked and relocated binary to flash ROM, and understands the relocation types in order to modify machine code instructions that need adjustment because of relocation.

Alternative Designs

The Contiki core symbol table contains all externally visible symbols in the Contiki core. Many of the symbols may never need to be accessed by loadable programs, thus causing ROM overhead. An alternative design would be to let the symbol table include only a handful of symbols, entry points, that define the only ways for an application program to interact with the core. This would lead to a smaller symbol table, but would also require a detailed specification of which entry points that should be included in the symbol table. The main reason why we did not chose this design, however, is that we wish to be able to replace modules at any level of the system. For this reason, we chose to provide the same amount of symbols to an application program as it would have, would it have been compiled directly into the core. However, we are continuing to investigate this alternative design for future versions of the system.

11.5.4 The Java Virtual Machine

We ported the Java virtual machine (JVM) from lejOS [8], a small operating system originally developed for the Lego Mindstorms. The Lego Mindstorms are equipped with an Hitachi H8 microcontroller with 32 kilobytes of RAM available for user programs such as the JVM. The lejOS JVM works within this constrained memory while featuring preemptive threads, recursion, synchronization and exceptions. The Contiki port required changes to the RAM-only model of the lejOS JVM. To be able to run Java programs within the 2 kilobytes of RAM available on our hardware platform, Java classes needs to be stored in flash ROM rather than in RAM. The Contiki port stores the class descriptions including bytecode in flash ROM memory. Static class data and class flags that denote if classes have been initialized are stored in RAM as well as object instances and execution stacks. The RAM requirements for the Java part of typical sensor applications are a few hundred bytes.

Java programs can call native code methods by declaring native Java methods. The Java virtual machine dispatches calls to native methods to native code. Any native function in Contiki may be called, including services that are part of a loaded Contiki program.

11.5.5 CVM - the Contiki Virtual Machine

We designed the Contiki Virtual Machine, CVS, to be a compromise between an application-specific and a generic virtual machine. CVM can be configured

for the application running on top of the machine by allowing functions to be either implemented as native code or as CVM code. To be able to run the same programs for the Java VM and for CVM, we developed a compiler that compiles a subset of the Java language to CVM bytecode.

The design of CVM is intentionally similar to other virtual machines, including Maté [19], VM* [18], and the Java virtual machine. CVM is a stack-based machine with separated code and data areas. The CVM instruction set contains integer arithmetic, unconditional and conditional branches, and method invocation instructions. Method invocation can be done in two ways, either by invocation of CVM bytecode functions, or by invocation of functions implemented in native code. Invocation of native functions is done through a special instruction for calling native code. This instruction takes one parameter, which identifies the native function that is to be called. The native function identifiers are defined at compile time by the user that compiles a list of native functions that the CVM program should be able to call. With the native function interface, it is possible for a CVM program to call any native functions provided by the underlying system, including services provided by loadable programs.

Native functions in a CVM program are invoked like any other function. The CVM compiler uses the list of native functions to translate calls to such functions into the special instruction for calling native code. Parameters are passed to native functions through the CVM stack.

11.6 Evaluation

To evaluate dynamic linking of native code we compare the energy costs of transferring, linking, relocating, loading, and executing a native code module in ELF format using dynamic linking with the energy costs of transferring, loading, and executing the same program compiled for the CVM and the Java virtual machine. We devise a simple model of the energy consumption of the reprogramming process. Thereafter we experimentally quantify the energy and memory consumption as well as the execution overhead for the reprogramming, the execution methods and the applications. We use the results of the measurements as input into the model which enables us to perform a quantitative comparison of the energy-efficiency of the reprogramming methods.

We use the ESB board [33] and the Telos Sky board [29] as our experimental platforms. The ESB is equipped with an MSP430 microcontroller with 2 kilobytes of RAM and 60 kilobytes of flash ROM, an external 64 kilobyte

```
PROCESS_THREAD(test_blink, ev, data)
{
  static struct etimer t;
  PROCESS_BEGIN();

  etimer_set(&t, CLOCK_SECOND);

  while(1) {
    leds_on(LEDS_GREEN);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));
    etimer_reset(&t);

    leds_off(LEDS_GREEN);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));
    etimer_reset(&t);
  }

  PROCESS_END();
}
```

Figure 11.3: Example Contiki program that toggles the LEDs every second.

EEPROM, as well as a set of sensors and a TR1001 radio transceiver. The Telos Sky is equipped with an MSP430 microcontroller with 10 kilobytes of RAM and 48 kilobytes of flash ROM together with a CC2420 radio transceiver. We use the ESB to measure the energy of receiving, storing, linking, relocating, loading and executing loadable modules and the Telos Sky to measure the energy of receiving loadable modules.

We use three Contiki programs to measure the energy efficiency and execution overhead of our different approaches. Blinker, the first of the two programs, is shown in Figure 11.3. It is a simple program that toggles the LEDs every second. The second program, Object Tracker, is an object tracking application based on abstract regions [35]. To allow running the programs both as native code, as CVM code, and as Java code we have implemented these programs both in C and Java. A schematic illustration of the C implementation is in Figure 11.4. To support the object tracker program, we implemented a subset of the abstract regions mechanism in Contiki. The Java and CVM versions of the program call native code versions of the abstract regions functions. The third program is a simple 8 by 8 vector convolution calculation.

```
PROCESS_THREAD(use_regions_process, ev, data)
{
  PROCESS_BEGIN();

  while(1) {

    value = pir_sensor.value();

    region_put(reading_key, value);
    region_put(reg_x_key, value * loc_x());
    region_put(reg_y_key, value * loc_y());
    if(value > threshold) {
      max = region_max(reading_key);

      if(max == value) {
        sum = region_sum(reading_key);
        sum_x = region_sum(reg_x_key);
        sum_y = region_sum(reg_y_key);
        centroid_x = sum_x / sum;
        centroid_y = sum_y / sum;
        send(centroid_x, centroid_y);
      }
    }

    etimer_set(&t, PERIODIC_DELAY);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));

  }

  PROCESS_END();
}
```

Figure 11.4: Schematic implementation of an object tracker based on abstract regions.

11.6.1 Energy Consumption

We model the energy consumption E of the reprogramming process with

$$E = E_p + E_s + E_l + E_f$$

where E_p is the energy spent in transferring the object over the network, E_s the energy cost of storing the object on the device, E_l the energy consumed by linking and relocating the object, and E_f the required energy for of storing the linked program in flash ROM. We use a simplified model of the network propagation energy where we assume a propagation protocol where the energy consumption E_p is proportional to the size of the object to be transferred. Formally,

$$E_p = P_p s_o$$

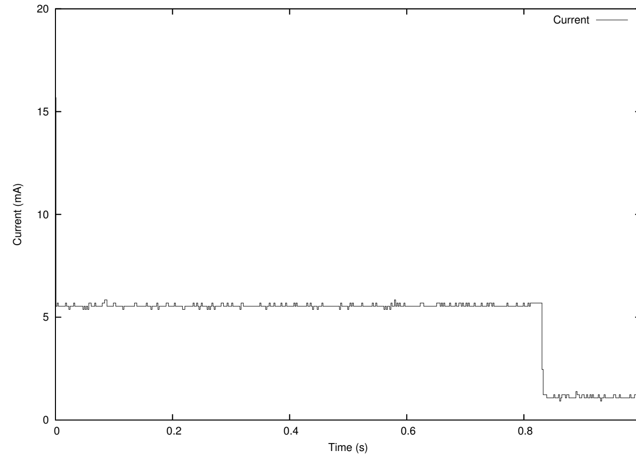

Figure 11.5: Current draw for receiving 1000 bytes with the TR1001.

where s_o is the size of the object file to be transfered and P_p is a constant scale factor that depends on the network protocol used to transfer the object. We use similar equations for E_s (energy for storing the binary) and E_l (energy for linking and relocating). The equation for E_f (the energy for loading the binary to ROM) contains the size of the compiled code size of the program instead of the size of the object file. This model is intentionally simple and we consider it good enough for our purpose of comparing the energy-efficiency of different reprogramming schemes.

Lower Bounds on Radio Reception Energy

We measured the energy consumption of receiving data over the radio for two different radio transceivers: the TR1001 [32], that is used on the ESB board, and the CC2420 [2], that conforms to the IEEE 802.15.4 standard [11] and is used on the Telos Sky board. The TR1001 provides a very low-level interface to the radio medium. The transceiver decodes data at the bit level and transmits the bits in real-time to the CPU. Start bit detection, framing, MAC layer, checksums, and all protocol processing must be done in software running on the CPU. In contrast, the interface provided by the CC2420 is at a higher level. Start bits, framing, and parts of the MAC protocol are handled by the transceiver. The software driver handles incoming and outgoing data on the

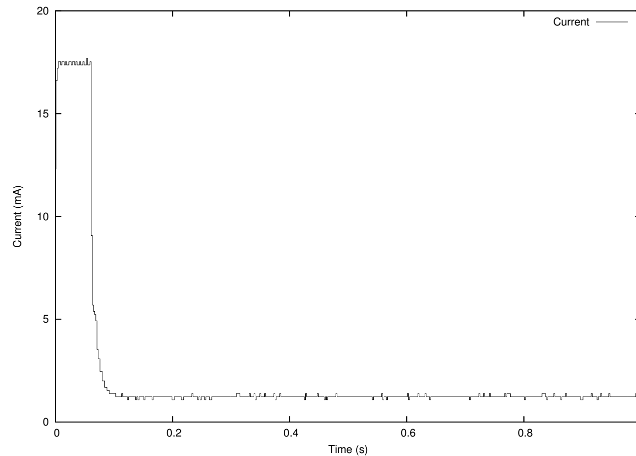

Figure 11.6: Current draw for receiving 1000 bytes with the CC2420.

packet level.

Since the TR1001 operates at the bit-level, the communication speed of the TR1001 is determined by the CPU. We use a data rate of 9600 bits per second. The CC2420 has a data rate of 250 kilobits per second, but also incurs some protocol overhead as it provides a more high-level interface.

Figures 11.5 and 11.6 show the current draw from receiving 1000 bytes of data with the TR1001 and CC2420 radio transceivers. These measurements constitute a lower bound on the energy consumption for receiving data over the radio, as they do not include any control overhead caused by a code propagation protocol. Nor do they include any packet headers. An actual propagation protocol would incur overhead because of both packet headers and control traffic. For example, the Deluge protocol has a control packet overhead of approximately 20% [14]. This overhead is derived from the total number of control packets and the total number of data packets in a sensor network. The average overhead in terms of number of excessive data packets received is 3.35 [14]. In addition to the actual code propagation protocol overhead, there is also overhead from the MAC layer, both in terms of packet headers and control traffic.

The TR1001 provides a low-level interface to the CPU, which enabled us to measure only the current draw of the receiver. We first measured the time required for receiving one byte of data from the radio. To produce the graph

Transceiver	Time (s)	Energy (mJ)	Time per byte (s)	Energy per byte (mJ)
TR1001	0.83	21	0.0008	0.021
CC2420	0.060	4.8	0.00006	0.0048

Table 11.2: Lower bounds on the time and energy consumption for receiving 1000 bytes with the TR1001 and CC2420 transceivers. All values are rounded to two significant digits.

in the figure, we measured the current draw of an ESB board which we had programmed to turn on receive mode and busy-wait for the time corresponding to the reception time of 1000 bytes.

When measuring the reception current draw of the CC2420, we could not measure the time required for receiving one byte because the CC2420 does not provide an interface at the bit level. Instead, we used two Telos Sky boards and programmed one to continuously send back-to-back packets with 100 bytes of data. We programmed the other board to turn on receive mode when the on-board button was pressed. The receiver would receive 1000 bytes of data, corresponding to 10 packets, before turning the receiver off. We placed the two boards next to each other on a table to avoid packet drops. We produced the graph in Figure 11.6 by measuring the current draw of the receiver Telos Sky board. To ensure that we did not get spurious packet drops, we repeated the measurement five times without obtaining differing results.

Table 11.2 shows the lower bounds on the time and energy consumption for receiving data with the TR1001 and CC2420 transceivers. The results show that while the current draw of the CC2420 is higher than that of the TR1001, the energy efficiency in terms of energy per byte of the CC2420 is better because of the shorter time required to receive the data.

Energy Consumption of Dynamic Linking

To evaluate the energy consumption of dynamic linking, we measure the energy required for the Contiki dynamic linker to link and load two Contiki programs. Normally, Contiki loads programs from the radio network but to avoid measuring any unrelated radio or network effects, we stored the loadable object files in flash ROM before running the experiments. The loadable objects were stored as ELF files from which all debugging information and symbols that were not needed for run-time linking was removed. At boot-up, one ELF file was copied

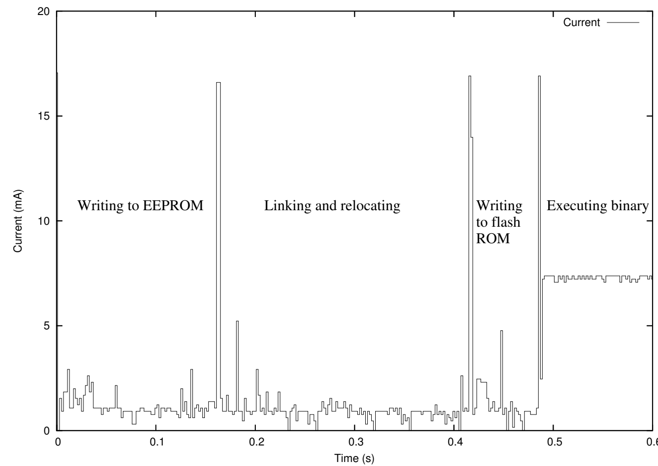

Figure 11.7: Current draw for writing the Blinker ELF file to EEPROM (0 - 0.166 s), linking and relocating the program (0.166 - 0.418 s), writing the resulting code to flash ROM (0.418 - 0.488 s), and executing the binary (0.488 s and onward). The current spikes delimit the three steps and are intentionally caused by blinking on-board LEDs. The high energy consumption when executing the binary is caused by the green LED.

into an on-board EEPROM from where the Contiki dynamic linker linked and relocated the ELF file before it loaded the program into flash ROM.

Figure 11.7 shows the current draw when loading the Blinker program, and Figure 11.8 shows the current draw when loading the Object Tracker program. The current spikes seen in both graphs are intentionally caused by blinking the on-board LEDs. The spikes delimit the four different steps that the loader is going through: copying the ELF object file to EEPROM, linking and relocating the object code, copying the linked code to flash ROM, and finally executing the loaded program. The current draw of the green LED is slightly above 8 mA, which causes the high current draw when executing the blinker program (Figure 11.7). Similarly, when the object tracking application starts, it turns on the radio for neighbor discovery. This causes the current draw to rise to around 6 mA in Figure 11.8, and matches the radio current measurements in Figure 11.5.

Table 11.3 shows the energy consumption of loading and linking the

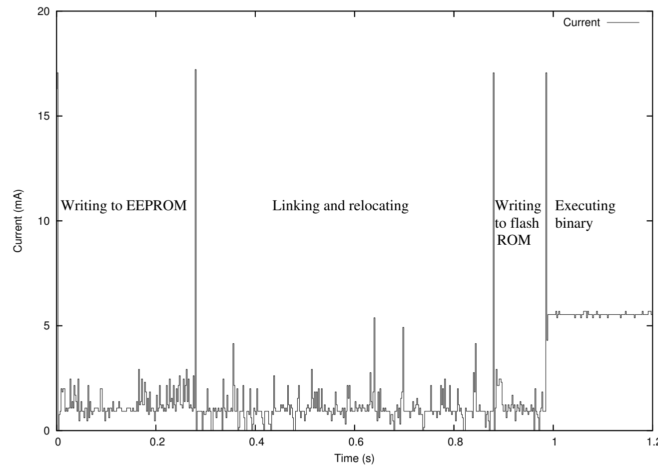

Figure 11.8: Current draw for writing the Object Tracker ELF file to EEPROM (0 - 0.282 s), linking and relocating the program (0.282 - 0.882 s), writing the resulting code to flash ROM (0.882 - 0.988 s), and executing the binary (0.988 s and onward). The current spikes delimit the three steps and are intentionally caused by blinking on-board LEDs. The high current draw when executing the binary comes from the radio being turned on.

Blinker program. The energy was obtained from integration of the curve from Figure 11.7 and multiplying it by the voltage used in our experiments (4.5 V). We see that the linking and relocation step is the most expensive in terms of energy. It is also the longest step.

To evaluate the energy overhead of the ELF file format, we compare the energy consumption for receiving four different Contiki programs using the ELF and CELF formats. In addition to the two programs from Figures 11.3 and 11.4 we include the code for the Contiki code propagation mechanism and a network publish/subscribe program that performs periodic flooding and converging of information. The two latter programs are significantly larger. We calculate an estimate of the required energy for receiving the files by using the measured energy consumption of the CC2420 radio transceiver and multiply it by the average overhead by the Deluge code propagation protocol, 3.35 [14]. The results are listed in Table 11.4 and show that radio reception is more energy consuming than linking and loading a program, even for a small program. Fur-

Step	Blinker time (s)	Energy (mJ)	Obj. Tr. time (s)	Energy (mJ)
Wrt. EEPROM	0.164	1.1	0.282	1.9
Link & reloc	0.252	1.2	0.600	2.9
Wrt. flash ROM	0.070	0.62	0.106	0.76
Total	0.486	2.9	0.988	5.5

Table 11.3: Measured energy consumption of the storing, linking and loading of the 1056 bytes large Blinker binary and the 1824 bytes large Object Tracker binary. The size of the Blinker code is 130 bytes and the size of the Object Tracker code is 344 bytes.

thermore, the results show that the relative average size and energy overhead for ELF files compared to the code and data contained in the files is approximately 4 whereas the relative CELF overhead is just under 2.

11.6.2 Memory Consumption

Memory consumption is an important metric for sensor nodes since memory is a scarce resource on most sensor node platforms. The ESB nodes feature only 2 KB RAM and 60 KB ROM while Mica2 motes provide 128 KB of program memory and 4 KB of RAM. The less memory required for reprogramming, the more is left for applications and support for other important tasks such as security which may also require a large part of the available memory [28].

Table 11.5 lists the memory requirements of the static linker, the dynamic linker and loader, the CVM and the Java VM. The dynamic linker needs to keep a table of all core symbols in the system. For a complete Contiki system with process management, networking, the dynamic loader, memory allocation, Contiki libraries, and parts of the standard C library, the symbol table requires about 4 kilobytes of ROM. This is included in the ROM size for the dynamic linker.

11.6.3 Execution Overhead

To measure the execution overhead of the application specific virtual machine and the Java virtual machine, we implemented the object tracking program in Figure 11.4 in C and Java. We compiled the Java code to CVM code and Java bytecode. We ran the compiled code on the MSP430-equipped ESB board.

Program	Code size	Data size	ELF file size	ELF file size overhead	ELF radio reception energy (mJ)
Blinker	130	14	1056	7.3	17
Object tracker	344	22	1668	5.0	29
Code propagator	2184	10	5696	2.6	92
Flood/converge	4298	42	8456	1.9	136

Program	Code size	Data size	CELF file size	CELF file size overhead	CELF radio reception energy (mJ)
Blinker	130	14	361	2.5	5.9
Object tracker	344	22	758	2.0	12
Code propagator	2184	10	3686	1.7	59
Flood/converge	4298	42	5399	1.2	87

Table 11.4: The overhead of the ELF and CELF file formats in terms of bytes and estimated reception energy for four Contiki programs. The reception energy is the lower bound of the radio reception energy with the CC2420 chip, multiplied by the average Deluge overhead (3.35).

The native C code was compiled with the MSP430 port of GCC version 3.2.3. The MSP430 digitally-controlled oscillator was set to clock the CPU at a speed of 2.4576 MHz. We measured the execution time of the three implementations using the on-chip timer A1 that was set to generate a timer interrupt 1000 times per second. The execution times are averaged over 5000 iterations of the object tracking program.

The results in Table 11.6 show the execution time of one run of the object tracking application from Figure 11.4. The execution time measurements are averaged over 5000 runs of the object tracking program. The energy consumption is calculated by multiplying the execution time with the average energy consumption when a program is running with the radio turned off. The table shows that the overhead of the Java virtual machine is higher than that of the CVM, which is turn is higher than the execution overhead of the native C code.

All three implementations of the tracker program use the same abstract regions library which is compiled as native code. Thus much of the execution time in the Java VM and CVM implementations of the object tracking program is spent executing the native code in the abstract regions library. Essentially, the virtual machine simply acts as a dispatcher of calls to various native func-

Module	ROM	RAM
Static loader	670	0
Dynamic linker, loader	5694	18
CVM	1344	8
Java VM	13284	59

Table 11.5: Memory requirements, in bytes. The ROM size for the dynamic linker includes the symbol table. The RAM figures do not include memory for programs running on top of the virtual machines.

Execution type	Execution time (ms)	Energy (mJ)
Native	0.479	0.00054
CVM	0.845	0.00095
Java VM	1.79	0.0020

Table 11.6: Execution times and energy consumption of one iteration of the tracking program.

tions. For programs that spend a significant part of their time executing virtual machine code the relative execution times are significantly higher for the virtual machine programs. To illustrate this, Table 11.7 lists the execution times of a convolution operation of two vectors of length 8. Convolution is a common operation in digital signal processing where it is used for algorithms such as filtering or edge detection. We see that the execution time of the program running on the virtual machines is close to ten times that of the native program.

11.6.4 Quantitative Comparison

Using our model from Section 11.6.1 and the results from the above measurements, we can calculate approximations of the energy consumption for distribution, reprogramming, and execution of native and virtual machine programs in order to compare the methods with each other. We set P_p, the scale factor of the energy consumption for receiving an object file, to the average Deluge overhead of 3.35.

Execution type	Execution time (ms)	Energy (mJ)
Native	0.67	0.00075
CVM	58.52	0.065
Java VM	65.6	0.073

Table 11.7: Execution times and energy consumption of the 8 by 8 vector convolution.

Step	Dynamic linking (mJ)	Full image replacement (mJ)
Receiving	17	330
Wrt. EEPROM	1.1	22
Link & reloc	1.4	-
Wrt. flash ROM	0.45	72
Total	20	424

Table 11.8: Comparison of energy-consumption of reprogramming the blinker application using dynamic linking with an ELF file and full image replacement methods.

Dynamic Linking vs Full Image Replacement

We first compare the energy costs for the two native code reprogramming models: dynamic linking and full image replacement. Table 11.8 shows the results for the energy consumption of reprogramming the blinker application. The size of blinker application including the operating system is 20 KB which is about 20 times the size of the blinker application itself. Even though no linking needs to be performed during the full image replacement, this method is about 20 times more expensive to perform a whole image replacement compared to a modular update using the dynamic linker.

Dynamic Linking vs Virtual Machines

We use the tracking application to compare reprogramming using the Contiki dynamic linker with code updates for the CVM and the Java virtual machine. CVM programs are typically very small and are not stored in EEPROM, nor are they linked or written to flash. Java uncompressed class files are loaded into flash ROM before they are executed. Table 11.9 shows the sizes of the

Step	ELF	CELF	CVM	Java
Size (bytes)	1824	968	123	1356
Receiving	29	12	2.0	22
Wrt. EEPROM	1.9	0.80	-	-
Link & reloc	2.5	2.5	-	-
Wrt. flash ROM	1.2	1.2	-	4.7
Total	35	16.5	2.0	26.7

Table 11.9: Comparison of energy-consumption in mJ of reprogramming for the object tracking application using the four different methods.

corresponding binaries and the energy consumption of each reprogramming step.

As expected, the process of updating sensor nodes with native code is less energy-efficient than updating with a virtual machine. Also, as shown in Table 11.6, executing native code is more energy-efficient than executing code for the virtual machines.

By combining the results in Table 11.6 and Table 11.9, we can compute break-even points for how often we can execute native code as opposed to virtual machine code for the same energy consumption. That is, after how many program iterations do the cheaper execution costs outweigh the more expensive code updates.

Figure 11.9 shows the modeled energy consumption for executing the Object Tracking program using native code loaded with an ELF object file, native code loaded with an CELF object file, CVM code, and Java code. We see that the Java virtual machine is expensive in terms of energy and will always require more energy than native code loaded with a CELF file. For native code loaded with an ELF file the energy overhead due to receiving the file makes the Java virtual machine more energy efficient until the program is repeated a few thousand times. Due to the small size of the CVM code it is very energy efficient for small numbers of program iterations. It takes about 40000 iterations of the program before the interpretation overhead outweigh the linking and loading overhead of same program running as native code and loaded as a CELF file. If the native program was loaded with an ELF file, however, the CVM program needs to be run approximately 80000 iterations before the energy costs are the same. At the break-even point, the energy consumption is only about one fifth of the energy consumption for loading the blink program

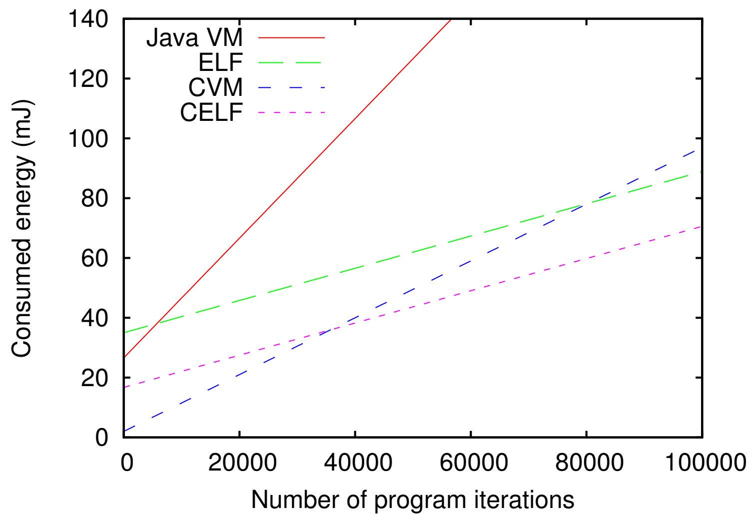

Figure 11.9: Break-even points for the object tracking program implemented with four different linking and execution methods.

using full image replacement as shown in Table 11.8.

In contrast with Figure 11.9, Figure 11.10 contains the break-even points from the vector convolution in Table 11.7. We assume that the convolution algorithm is part of a program with the same size as in Figure 11.9 so that the energy consumption for reprogramming is the same. In this case the break-even points are drastically lower than in Figure 11.9. Here the native code loaded with an ELF file outperforms the Java implementation already at 100 iterations. The CVM implementation has spent as much energy as the native ELF implementation after 500 iterations.

11.6.5 Scenario Suitability

We can now apply our results to the software update scenarios discussed in Section 11.2. In a scenario with frequent code updates, such as the dynamic application scenario or during software development, a low loading overhead is to prefer. From Figure 11.9 we see that both an application-specific virtual machine and a Java machine may be good choices. Depending on the type of

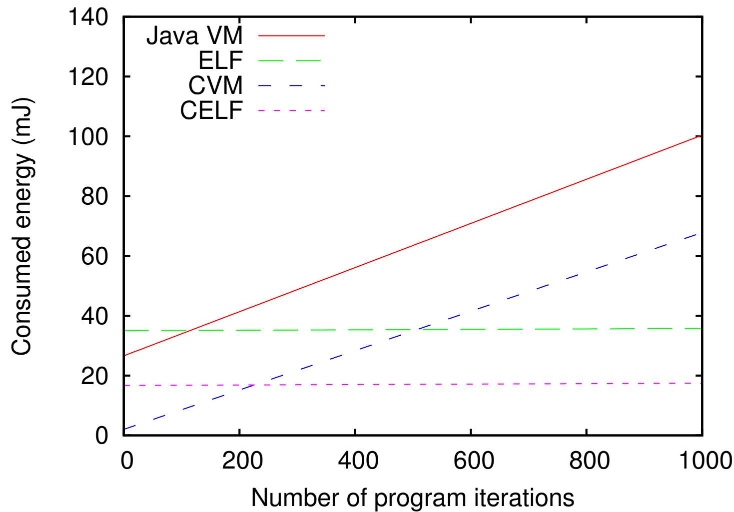

Figure 11.10: Break-even points for the vector convolution implemented with four different linking and execution methods.

application it may be beneficial to decide to run the program on top of a more flexible virtual machine such as the Java machine. The price for such a decision is higher energy overhead.

In scenarios where the update frequency is low, e.g. when fixing bugs in installed software or when reconfiguring an installed application, the higher price for dynamic linking may be worth paying. If the program is continuously run for a long time, the energy savings of being able to use native code outweigh the energy cost of the linking process. Furthermore, with a virtual machine it may not be possible to make changes to all levels of the system. For example, a bug in a low-level driver can usually only be fixed by installing new native code. Moreover, programs that are computationally heavy benefit from being implemented as native code as native code has lower energy consumption than virtual machine code.

The results from Figures 11.9 and 11.10 suggest that a combination of virtual machine code and native code can be energy efficient. For many situations this may be a viable alternative to running only native code or only virtual machine code.

Module	Lines of code, total	Lines of code, relocation function
Generic linker	292	
MSP430-specific	45	8
AVR-specific	143	104

Table 11.10: Number of lines of code for the dynamic linker and the microcontroller-specific parts.

11.6.6 Portability

Because of the diversity of sensor network platforms, the Contiki dynamic linker is designed to be portable between different microcontrollers. The dynamic linker is divided into two modules: a generic part that parses and analyzes the ELF/CELF that is to be loaded, and a microcontroller-specific part that allocates memory for the program to be loaded, performs code and data relocation, and writes the linked program into memory.

To evaluate the portability of our design we have ported the dynamic linker to two different microcontrollers: the TI MSP430 and the Atmel AVR. The TI MSP430 is used in several sensor network platforms, including the Telos Sky and the ESB. The Atmel AVR is used in the Mica2 motes.

Table 11.10 shows the number of lines of code needed to implement each module. The dramatic difference between the MSP430-specific module and the AVR-specific module is due to the different addressing modes used by the machine code of the two microcontrollers. While the MSP430 has only one addressing mode, the AVR has 19 different addressing modes. Each addressing mode must be handled differently by the relocation function, which leads to a larger amount of code for the AVR-specific module.

11.7 Discussion

Standard file formats. Our main motivation behind choosing the ELF format for dynamic linking in Contiki was that the ELF format is a standard file format. Many compilers and utilities, including all GCC utilities, are able to produce and handle ELF files. Hence no special software is needed to compile and upload new programs into a network of Contiki nodes. In contrast, FlexCup [27] or diff-based approaches require the usage of specially crafted

utilities to produce meta data or diff scripts required for uploading software. These special utilities also need to be maintained and ported to the full range of development platforms used for software development for the system.

Operating system support. Dynamic linking of ELF files requires support from the underlying operating system and cannot be done on monolithic operating systems such as TinyOS. This is a disadvantage of our approach. For monolithic operating systems, an approach such as FlexCup is better suited.

Heterogeneity. With diff-based approaches a binary diff is created either at a base station or by an outside server. The server must have knowledge of the exact software configuration of the sensor nodes on which the diff script is to be run. If sensor nodes are running different versions of their software, diff-based approaches do not scale.

Specifically, in many of our development networks we have witnessed a form of *micro heterogeneity* in the software configuration. Many sensor nodes, which have been running the exact same version of the Contiki operating system, have had small differences in the address of functions and variables in the core. This micro heterogeneity comes from the different core images being compiled by different developers, each having slightly different versions of the C compiler, the C library and the linker utilities. This results in small variations of the operating system image depending on which developer compiled the operating system image. With diff-based approaches micro heterogeneity poses a big problem, as the base station would have to be aware of all the small differences between each node.

Combination of native and virtual machine code. Our results suggest that a combination of native and virtual machine code is an energy efficient alternative to pure native code or pure virtual machine code approaches. The dynamic linking mechanism can be used to load the native code that is used by the virtual machine code by the native code interfaces in the virtual machines.

11.8 Related Work

Because of the importance of dynamic reprogramming of wireless sensor networks there has been a lot of effort in the area of software updates for sensor nodes both in the form of system support for software updates and execution environments that directly impact the type and size of updates as well as distribution protocols for software updates.

Mainwaring et al. [26] also identified the trade-off between using virtual machine code that is more expensive to run but enables more energy-efficient

updates and running native code that executes more efficiently but requires more costly updates. This trade-off has been further discussed by Levis and Culler [19] who implemented the Maté virtual machine designed to both simplify programming and to leverage energy-efficient large-scale software updates in sensor networks. Maté is implemented on top of TinyOS.

Levis and Culler later enhanced Maté by application specific virtual machines (ASVMs) [20]. They address the main limitations of Maté: flexibility, concurrency and propagation. Whereas Maté was designed for a single application domain only, ASVM supports a wide range of application domains. Further, instead of relying on broadcasts for code propagation as Maté, ASVM uses the trickle algorithm [21].

The MagnetOS [23] system uses the Java virtual machine to distribute applications across an ad hoc network of laptops. In MagnetOS, Java applications are partitioned into distributed components. The components transparently communicate by raising events. Unlike Maté and Contiki, MagnetOS targets larger platforms than sensor nodes such as PocketPC devices. SensorWare [1] is another script-based proposal for programming nodes that targets larger platforms. VM* is a framework for runtime environments for sensor networks [18]. Using this framework Koshy and Pandey have implemented a subset of the Java Virtual Machine that enables programmers to write applications in Java, and access sensing devices and I/O through native interfaces.

Mobile agent-based approaches extend the notion of injected scripts by deploying dynamic, localized and intelligent mobile agents. Using mobile agents, Fok et al. have built the Agilla platform that enables continuous reprogramming by injecting new agents into the network [9].

TinyOS uses a special description language for composing a system of smaller components [10] which are statically linked with the kernel to a complete image of the system. After linking, modifying the system is not possible [19] and hence TinyOS requires the whole image to be updated even for small code changes.

Systems that offer loadable modules besides Contiki include SOS [12] and Impala [24]. Impala features an application updater that enables software updates to be performed by linking in updated modules. Updates in Impala are coarse-grained since cross-references between different modules are not possible. Also, the software updater in Impala was only implemented for much more resource-rich hardware than our target devices. The design of SOS [12] is very similar to the Contiki system: SOS consists of a small kernel and dynamically-loaded modules. However, SOS uses position independent code to achieve relocation and jump tables for application programs to access the operating

system kernel. Application programs can register function pointers with the operating system for performing inter-process communication. Position independent code is not available for all platforms, however, which limits the applicability of this approach.

FlexCup [27] enables run-time installation of software components in TinyOS and thus solves the problem that a full image replacement is required for reprogramming TinyOS applications. In contrast to our ELF-based solution, FlexCup uses a non-standard format and is less portable. Further, FlexCup requires a reboot after a program has been installed, requiring an external mechanism to save and restore the state of all other applications as well as the state of running network protocols across the reboot. Contiki does not need to be rebooted after a program has been installed.

FlexCup also requires a complete duplicate image of the binary image of the system to be stored in external flash ROM. The copy of the system image is used for constructing a new system image when a new program has been loaded. In contrast, the Contiki dynamic linker does not alter the core image when programs are loaded and therefore no external copy of the core image is needed.

Since the energy consumption of distributing code in sensor networks increases with the size of the code to be distributed several attempts have been made to reduce the size of the code to be distributed. Reijers and Langendoen [31] produce an edit script based on the difference between the modified and original executable. After various optimizations including architecture-dependent ones, the script is distributed. A similar approach has been developed by Jeong and Culler [15] who use the rsync algorithm to generate the difference between modified and original executable. Koshy and Pandey's diff-based approach [17] reduces the amount of flash rewriting by modifying the linking procedure so that functions that are not changed are not shifted.

XNP [16] was the previous default reprogramming mechanism in TinyOS which is used by the multi-hop reprogramming scheme MOAP (Multi-hop Over-the-Air Programming) developed to distribute node images in the sensor network. MOAP distributes data to a selective number of nodes on a neighbourhood-by-neighbourhood basis that avoids flooding [34]. In Trickle [21] virtual machine code is distributed to a network of nodes. While Trickle is restricted to single packet dissemination, Deluge adds support for the dissemination of large data objects [14].

11.9 Conclusions

We have presented a highly portable dynamic linker and loader that uses the standard ELF file format and compared the energy-efficiency of run-time dynamic linking with an application specific virtual machine and a Java virtual machine. We show that dynamic linking is feasible even for constrained sensor nodes.

Our results also suggest that a combination of native and virtual machine code provide an energy efficient alternative to pure native code or pure virtual machine approaches. The native code that is called from the virtual machine code can be updated using the dynamic linker, even in heterogeneous systems.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European Commission under contract IST-004536-RUNES. Thanks to our paper shepherd Feng Zhao for reading and commenting on the paper.

Bibliography

[1] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, San Francisco, USA, May 2003.

[2] Chipcon AS. CC2420 Datasheet (rev. 1.3), 2005.
URL: http://www.chipcon.com/

[3] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[4] H. Dai, M. Neufeld, and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys*, pages 176–187, 2004.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE Emnets '04)*, Tampa, Florida, USA, November 2004.

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006*, Boulder, Colorado, USA, 2006.

[7] D. Estrin (editor). *Embedded everywhere: A research agenda for networked systems of embedded computers*. National Academy Press, 1st edition, October 2001. ISBN: 0309075688

[8] G. Ferrari, J. Stuber, A. Gombos, and D. Laverde, editors. *Programming Lego Mindstorms with Java with CD-ROM*. Syngress Publishing, 2002. ISBN: 1928994555

[9] C. Fok, G. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, Tokyo, Japan, June 2005.

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, San Diego, California, USA, June 2003.

[11] J. A. Gutierrez, M. Naeve, E. Callaway, M. Bourgeois, V. Mitter, and B. Heile. IEEE 802.15.4: A developing standard for low-power low-cost wireless personal area networks. *IEEE Network*, 15(5):12–19, September/October 2001.

[12] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys '05)*, Seattle, WA, USA, June 2005.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusets, USA, November 2000.

[14] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys '04)*, Baltimore, Maryland, USA, November 2004.

[15] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON (2004)*, Santa Clara, California, USA, October 2004.

[16] J. Jeong, S. Kim, and A. Broad. Network reprogramming. TinyOS documentation, 2003. Visited 2006-04-06.
URL: http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf

[17] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Wireless Sensor Networks*, 2005.

[18] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *Proc. SenSys'05*, San Diego, CA, USA, November 2005.

[19] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[20] P. Levis, D. Gay, and D Culler. Active sensor networks. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005.

[21] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'04)*, March 2004.

[22] J. Lilius and I. Paltor. Deeply embedded python, a virtual machine for embedded systems. Web page. 2006-04-06.
URL: http://www.tucs.fi/magazin/output.php?ID=2000.N2.LilDeEmPy

[23] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. Gün Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys*, pages 149–162, 2005.

[24] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proc. Second Intl. Conference on Mobile Systems, Applications and Services (MOBISYS 2004)*, June 2004.

[25] G. Mainland, L. Kang, S. Lahaie, D. C. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. In *Proceedings of the 2004 SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[26] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.

[27] P. José Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *European Workshop on Wireless Sensor Networks*, Zurich, Switzerland, January 2006.

[28] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: security protocols for sensor netowrks. In *Mobile Computing and Networking*, pages 189–199, 2001.

[29] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

[30] N. Ramanathan, E. Kohler, and D. Estrin. Towards a debugging system for sensor networks. *International Journal for Network Management*, 3(5), 2005.

[31] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, San Diego, CA, USA, September 2003.

[32] RF Monolithics. 868.35 MHz Hybrid Transceiver TR1001, 1999.
URL: http://www.rfm.com

[33] J. Schiller, H. Ritter, A. Liers, and T. Voigt. Scatterweb - low power nodes and energy aware routing. In *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, USA, January 2005.

[34] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[35] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of ACM/Usenix Networked Systems Design and Implementation (NSDI'04)*, San Francisco, California, USA, March 2004.

[36] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

SICS Dissertation Series

01: Bogumil Hausman, Pruning and Speculative Work in OR-Parallel PROLOG, 1990.

02: Mats Carlsson, Design and Implementation of an OR-Parallel Prolog Engine, 1990.

03: Nabiel A. Elshiewy, Robust Coordinated Reactive Computing in SANDRA, 1990.

04: Dan Sahlin, An Automatic Partial Evaluator for Full Prolog, 1991.

05: Hans A. Hansson, Time and Probability in Formal Design of Distributed Systems, 1991.

06: Peter Sjödin, From LOTOS Specifications to Distributed Implementations, 1991.

07: Roland Karlsson, A High Performance OR-parallel Prolog System, 1992.

08: Erik Hagersten, Toward Scalable Cache Only Memory Architectures, 1992.

09: Lars-Henrik Eriksson, Finitary Partial Inductive Definitions and General Logic, 1993.

10: Mats Björkman, Architectures for High Performance Communication, 1993.

11: Stephen Pink, Measurement, Implementation, and Optimization of Internet Protocols, 1993.

12: Martin Aronsson, GCLA. The Design, Use, and Implementation of a Program Development System, 1993.

13: Christer Samuelsson, Fast Natural-Language Parsing Using Explanation-Based Learning, 1994.

14: Sverker Jansson, AKL - - A Multiparadigm Programming Language, 1994.

15: Fredrik Orava, On the Formal Analysis of Telecommunication Protocols, 1994.

16: Torbjörn Keisu, Tree Constraints, 1994.

17: Olof Hagsand, Computer and Communication Support for Interactive Distributed Applications, 1995.

18: Björn Carlsson, Compiling and Executing Finite Domain Constraints, 1995.

19: Per Kreuger, Computational Issues in Calculi of Partial Inductive Definitions, 1995.

20: Annika Waern, Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction, 1996.

21: Björn Gambäck, Processing Swedish Sentences: A Unification-Based Grammar and Some Applications, 1997.

22: Klas Orsvärn, Knowledge Modelling with Libraries of Task Decomposition Methods, 1996.

23: Kia Höök, A Glass Box Approach to Adaptive Hypermedia, 1996.

24: Bengt Ahlgren, Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption, 1997.

25: Johan Montelius, Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, 1997.

26: Jussi Karlgren, Stylistic experiments in information retrieval, 2000.

27: Ashley Saulsbury, Attacking Latency Bottlenecks in Distributed Shared Memory Systems, 1999.

28: Kristian Simsarian, Toward Human Robot Collaboration, 2000.

29: Lars-åke Fredlund, A Framework for Reasoning about Erlang Code, 2001.

30: Thiemo Voigt, Architectures for Service Differentiation in Overloaded Internet Servers, 2002.

31: Fredrik Espinoza, Individual Service Provisioning, 2003.

32: Lars Rasmusson, Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design, 2002.

33: Martin Svensson, Defining, Designing and Evaluating Social Navigation, 2003.

34: Joe Armstrong, Making reliable distributed systems in the presence of software errors, 2003.

35: Emmanuel Frécon, DIVE on the Internet, 2004.

36: Rickard Cöster, Algorithms and Representations for Personalised Information Access, 2005.

37: Per Brand, The Design Philosophy of Distributed Programming Systems: the Mozart Experience, 2005.

38: Sameh El-Ansary, Designs and Analyses in Structured Peer-to-Peer Systems, 2005.

39: Erik Klintskog, Generic Distribution Support for Programming Systems, 2005.

40: Markus Bylund, A Design Rationale for Pervasive Computing User Experience, Contextual Change, and Technical Requirements, 2005.

41: Åsa Rudström, Co-Construction of hybrid spaces, 2005.

42: Babak Sadighi Firozabadi, Decentralised Privilege Management for Access Control, 2005.

43: Marie Sjölinder, Age-related Cognitive Decline and Navigation in Electronic Environments, 2006.

44: Magnus Sahlgren, The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations between Words in High-dimensional Vector Spaces, 2006.

45: Ali Ghodsi, Distributed k-ary System: Algorithms for Distributed Hash Tables, 2006.

46: Stina Nylander, Design and Implementation of Multi-Device Services, 2007

47: Adam Dunkels, Programming Memory-Constrained Networked Embedded Systems, 2007.